Universitetet i Oslo
Institutt for informatikk

Hierarkisk modellering og sanntidsvisualisering av skog

Kristian Børresen

12. november 2003

Denne oppgaven er til tittelen Cand.Scient. ved Institutt for Informatikk, Universitetet i Oslo. Jeg ønsker å først og fremt takke hovedveileder Morten Dæhlen og delveileder Thomas Engh Sevaldrud for god hjelp og tilbakemelding under arbeidet. Videre ønsker jeg å takke de som har satt av tid til å lese igjennom oppgaven og gitt tilbakemelding: Vera Louise Hauge, Per-Idar Evensen, Anette Gjetnes, Morten Berg og ikke minst Grete Hope.

Innhold

Kapittel 1

Innledning

Terrengvisualisering med naturtro gjengivelse har vært en kontinuerlig utfordring i datagrafikk. Den komplekse geometrien som finnes i naturen gjør at tilsvarende modeller i datamaskinen også blir komplekse. De fine detaljene, interaksjonen mellom ulike objekter og skyggeleggingen i naturen er alle faktorer som kompliserer representasjonen av naturen i en datamaskin.

I denne oppgaven ønsker vi å konsentrere oss om problemstillingen rundt visualiseringen av skog. Vi søker en modell som virkeliggjør opptegningen av flere tusen trær i sanntid. Innenfor datagrafikk betyr sanntidsvisualisering at opptegningen av bilderammer per sekund aldri må komme under et visst minimum. I denne oppgaven arbeider vi med 60 bilderammer per sekund som det minste tallet på hvor rask opptegningen må være. Lav oppdateringsrate vil føre til merkbar forsinkelse mellom bruker og en applikasjon.

Bruksområdet for modellen vår kan være dataspill, arkitekturvisualisering og flysimulatorer.

1.1 Beslektet arbeid

Visualiseringen av vegetasjon og trær har vært et aktivt forskningsfelt innenfor datagrafikk. Vi kan hovedsaklig dele dette feltet i to: de som er opptatt av modellering og distribusjon av vegetasjon for ulike økosystemer og de som er opptatt av naturtro gjengivelse innenfor sanntidsvisualisering. Særlig har visualiseringen av trær vært en viktig problemstilling.

Trær er, geometrisk sett, svært kompliserte objekter. En metode for å forenkle trevisualiseringen er å dele et tre i to separate deler: stammen/greiner og blader/nåler. Trestammen og greinene kan representeres ved polygoner. Blader og nåler derimot representeres ved flere gjennomskinnelige plan som tekstureres med bilderepresentasjoner [1], [2].

Bilderepresentasjoner av trær er en populær teknikk innenfor trevisualisering fordi den koster svært lite med hensyn på grafikk-prosessering. Det finnes metoder for å bytte ut bilderepresentasjoner med mer detaljerte polygon-

modeller etterhvert som treet blir større på skjermen [3], [4].

Rammeverket rundt vår modell er bygget på tidligere arbeid rundt terrengvisualisering. Datastrukturene vi bruker har vist seg å være svært effektive for å visualisere naturtro landskap i sanntid [5], [6], [7], [8].

1.2 Mål med oppgaven

I denne oppgaven har vi følgende mål:

- *Modellen skal genereres ved oppstart og under kjøringen av applikasjonen:* All data utenom teksturer som skogmodellen består av skal enten genereres i en preprosess eller dynamisk under kjøringen av applikasjon.
- *Skogmodellen skal være bygget opp som et hierarki:* Vi søker en modell som har hierarkisk struktur. Med dette så mener vi en modell som inneholder ulik detaljgrad for sine data. Dette har mange fordeler under visualisering.
- *Modellen skal være fleksibel:* Vi søker en skogmodell som er fleksibel i den forstand at den skal kunne integreres med en rekke ulike terreng.
- *Skogmodellen skal være lett å konfigurere:* Med lett å konfigurere mener vi at brukeren av applikasjonen enkelt skal kunne endre parametre som brukes til å bygge opp skogmodellen. Det skal også være enkelt å legge til nye tremodeller for bruk under visualiseringen.
- *Modellen skal visualiseres i sanntid:* Vi søker sanntidsvisualisering av skogmodellen med de fordeler og ulemper som dette medfører. Dette krever at visualiseringen må være rask og enkel.

1.3 Hvordan

For å nå våre mål, må vi foreta noen antagelser. Disse hjelper oss med å foreta avskjæringer som har liten innflytelse på det visuelle resultatet i modellen. Vi gir så en kort innføring i hvordan skogmodellen er bygget opp.

1.3.1 Antagelser

Den første antagelsen vi gjør går ut på at hvis et menneske befinner seg høyt over en skog er det tilhørende plantelivet i skogbunnen vanskelig å oppdage for det menneskelige øye. Busker, blomster og andre planter er som regel dekket over av trær. Det er først når man befinner seg under trærne at man begynner å legge merke til det mangfoldige plantelivet. Dermed kan vi

forenkle vår skogmodell ved å si at en skog i vårt tilfelle kun består av en mengde trær.

Den neste antagelsen er kun en utvidelse av den første. Fra et fugleperspektiv har nemlig det menneskelige øye også vanskeligheter med å skille individuelle trær fra hverandre. Er vi langt nok unna en skog, vil vi bare se "noe grønt". Ved å benytte oss av dette, kan vi unngå å tegne trær når vi befinner oss tilstrekkelig langt nok unna et skogsområde.

Selv om to trær av samme type ikke er like, har vi mennesker vanskeligheter med å skille mellom dem. Det menneskelige øye er ikke i stand til å skille alle de små detaljene ved et enkelt tre fra hverande [1]. Utnytter vi dette kan vi bygge opp vår skogsmodell ved hjelp av et lite antall ulike tretyper. Gir vi hvert enkelt tre små endringer i forhold til rotasjon og høyde, kan vi skape et realistisk bilde av en skog.

1.3.2 Oppbygning

For å generere en skogmodell som skal kunne visualiseres i sanntid, må vi bruke datastrukturer og teknikker som egner seg godt til dette formålet. Prosessen med å bygge opp skogmodellen kan deles inn i to deler:

- *Utplassering av treposisjoner i et landskap:* Vi har i denne oppgaven brukt et quadtre som datastruktur for å inndele et landskap i mindre og mer håndterlig biter. Utplasseringen av trær foregår ved å forbinde et gitt antall treposisjoner til hver node i quadtreet.

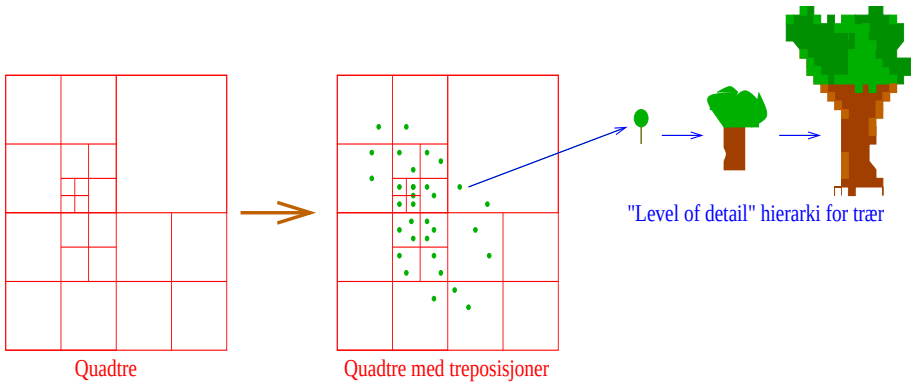
- *Ulike trerepresentasjoner:* De ulike trærne i landskapet kan deles inn i ulik detaljgrad. Trær langt unna behøver en enklere representasjon enn trær nærme kamera.

Denne inndelingen er vist på figur 1.1. Quadtreet deler inn et rektangulært (i vårt tilfelle kvadratisk) område i mindre områder jo dypere man traverserer det. Vi knytter treposisjoner til quadtreet ved å utplassere dem i landskapet der hvor terrenginformasjonen tilsier at vi har skog. Treposisjonene er organisert slik at vi under visualiseringen ser flere og flere trær jo dypere ned i quadtreet vi traverserer.

De individuelle trærne i landskapet består av ulike detaljerte representasjoner. Under kjøring velges det til enhver tid passende detaljgrad ut i fra et sett med gitte kriterier. De ulike trerepresentasjonene sies å være ordnet i et "Level-of-Detail" hierarki.

1.4 Organisering av oppgaven

Vi begynner med å se på hvordan moderne datagrafikksystemer er satt sammen, før vi omhandler et programvaregrensesnitt mot slike systemer kalt



Figur 1.1: *Oppbygningen av skogmodellen. Vi bruker et quadtre for å organisere landskapet vårt i rommet. Vi knytter treposisjoner til quadtreet for å representere skogen vår. Hvert tre i landskapet består av flere ulike detaljerte representasjoner.*

OpenGL. Etter dette gir vi en kort innføring i objektorientering og terminologien knyttet til denne. Dette er tema i kapittel 2.

Kapittel 3 introduserer de ulike modellene og datastrukturene som vi har brukt i denne oppgaven. Vi begynner med å introdusere egenskapene til quadtreet og hvorfor denne er effektiv i visualiseringssammenheng. Til slutt introduserer vi "Level-of-Detail" konseptet.

Utplassering av treposisjoner i terrenget er temaet for kapittel 4. Vi begynner med å introdusere datasettet som gir oss grunnlaget for å vite hvor i terrenget vi kan utplassere trær. Vi ser så på hvordan de utplasserte treposisjonene knyttes til quadtreet. Til slutt introduserer vi muligheten for å utplassere ulike tretyper før vi viser hvordan quadtreet kan genereres dynamisk under kjøring.

Kapittel 5 omhandler hvordan vi visualiserer trær i modellen. Vi introduserer et "Level-of-Detail" hierarki for alle gitte tretyper i skogsmodellen før vi viser hvordan vi effektivt kan ordne informasjonen på en fornuftig måte. Siden vi har kun gitt treposisjoner må vi til slutt vise hvordan vi kan knytte denne posisjonen til våre trerepresentasjoner.

Visualiseringsmetoder er temaet for kapittel 6. Vi tar for oss to ulike visualiseringsmetoder. Til slutt illustrerer vi hvordan vi kan knytte høydeverdier til vår modell.

Kapittel 7 måler resultatet av vår implementasjon. Vi måler ytelsen til skogmodellen med hvor mange trær vi klarer å tegne til skjerm. Videre arbeid og noen konklusjoner diskuteres i kapittel 8.

Kapittel 2

Bakgrunn

Datagrafikksystemer har endret seg radikalt de siste årene. Egen dedikert maskinvare for opptegning av tredimensjonal data i sanntid har blitt mer og mer vanlig i de fleste moderne datamaskiner. Slik maskinvare består av et grafikkort kalt en grafikkakselerator som er tilpasset for å håndtere store mengder med tredimensjonal data.

OpenGL er en programvarespesifikasjon som forenkler kommunikasjonen mellom en applikasjon og grafikkakseleratoren. Ved å gi ulike OpenGL instruksjoner i et program, vil disse bli sendt videre ned til grafikkakseleratoren for prosessering.

For å hjelpe oss med å organisere våre datastrukturer bruker vi objektorientering. Objektorientert programmering sørger for at skillet mellom konsept og implementasjon minskes. Dette gjøres ved å innføre abstrakte objekter.

2.1 Sanntidsvisualisering

Når vi arbeider med sanntidsvisualisering kan vi tenke oss at denne prosessen består av tre ulike faser som vist på figur 2.1 Vi tenker oss at dataflyten går fra applikasjonsfasen gjennom geometrifasen til rasteriseringsfasen, se figur 2.1. De tre fasene skisserer grovt hvordan moderne grafikksystemer er bygget opp [9, side 10].

Figur 2.1: *De tre fasene som tilsammen danner et moderne grafikksystem.*

Hastigheten på dataflyten gjennom de tre fasene på figur 2.1 bestemmer opptegningshastigheten. Opptegningshastigheten er ofte målt i bilderammer per sekund. Vi minner om at for å oppnå sanntidsvisualisering må vi

forhindre at opptegningshastigheten blir for lav.

2.1.1 Applikasjonsfasen

Målet med applikasjonsfasen er å lage en applikasjon som genererer tredimensjonale data som sendes videre til geometrifasen. Applikasjonsfasen er den eneste av de tre fasene på figur 2.1 som er fullstendig implementert i programvare. Dette vil si at utvikleren av applikasjonen har full kontroll over implementasjonen og kan optimalisere deretter.

Applikasjonsfasen genererer grafiske primitiver: enkle geometriske objekter som for eksempel punkt, linjestykker og trekanter. Jo færre grafiske primitiver en applikasjon genererer, desto raskere gjennomstrømning vil vi få gjennom grafikksystemet. Dette fører igjen til raskere opptegningshastighet. Det er dermed vanlig i applikasjonsfasen å bruke ulike algoritmer for å minske antallet grafiske primitiver uten at det går på bekostning av kvaliteten på det som til slutt blir bildet på skjermen. Andre viktige oppgaver til applikasjonsfasen er å prosessere data fra for eksempel tastatur eller mus.

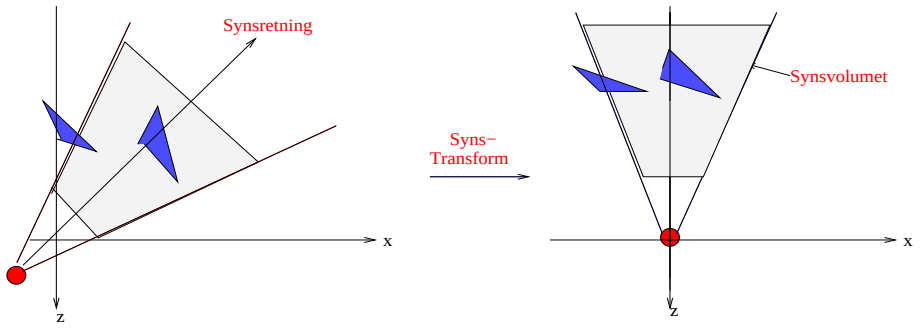
2.1.2 Geometrifasen

Geometrifasen har som oppgave å behandle grafiske primitiver. Denne jobben gjøres som nevnt i innledningen via egen dedikert maskinvare kalt en grafikkakselerator. Vi ønsker at grafikkakseleratoren skal klare å prosessere mange grafiske primitiver i sanntid, så slike grafikkort er nesten alltid implementert som en pipeline. De ulike trinnene i pipelinen er vist på figur 2.2.

Figur 2.2: *Moderne grafikkakseleratorer er bygget som en pipeline med hovedsaklig fem subprosesser: Modell og synstransform, Lyssetting, Projeksjon, Klipping og Skjermavbildning.*

En pipeline deler en prosess inn i flere subprosesser. Fordelen med dette er at vi ved å dele opp en prosess i n subprosesser generelt får en hastighetsøkning på faktor n. Dette forklarer hvorfor pipelinearkitekturen er så populær. Ulempen med en pipeline struktur er at subprosessen som tar lengst tid vil også være den subprosessen som bestemmer hvor rask pipelinen er.

Alle tredimensjonale modeller vi jobber med i datagrafikk eksisterer i sitt eget lokale koordinatsystem. Dette koordinatsystemet kalles for modellrommet. Vi må transformere de lokale koordinatene til et felles koordinatsystem kalt verdenskoordinater. Transformen som sørger for at alle objekter deler felles koordinatsystem kalles modell transformen.



Figur 2.3: *Synstransformen transformerer kameraet, som ligger i verdenskoordinater, til å ligge i origo med synsretningen langs den negative z-aksen. Dette koordinatsystemet kalles øyerommet. Vi ser også at kameraet har definert et eget synsvolum som bestemmer hva som er synlig for kameraet. Eksemplet viser synsvolumet til en perspektiv projeksjon.*

For å vite hva som er synlig på skjermen innenfor verdenskoordinatrommet, har vi innen datagrafikk gitt et virtuelt kamera. Kameraet ligger også i verdenskoordinatsystemet med en gitt synsretning som på figur 2.3. Synstransformen transformerer det virtuelle kameraet slik at det ligger i origo med synsretningen langs den negative z-aksen. Etter synstransformen er fullført ligger kameraet i øyerommet.

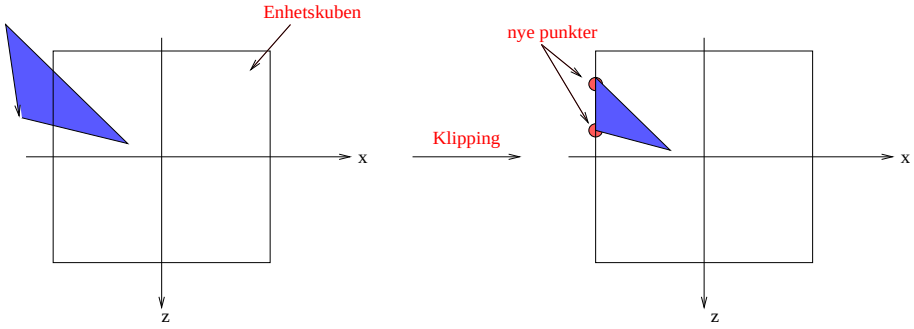
Begge transformene er implementert som en egen 4×4 matrise. For å effektivisere transformeringsprosessen konkatenerer vi begge transformene sammen til en matrise. Vi transformerer dermed direkte til øyerommet.

Neste steg i pipelinen er lyssetting. For å få en mer tredimensjonal følelse av objektene kan vi velge å skyggelegge dem avhengig av en eller flere lyskilder. Velger vi å skyggelegge et objekt må vi beregne den nye fargen til objektet avhengig av hvor lyskilden befinner seg i forhold til objektet.

Etter dette kan vi foreta en projeksjon. Det virtuelle kameraet definerer et synsvolum som spesifiserer hva som er synlig for kameraet. Projeksjonen forander synsvolumet til en enhetskube med ekstremalpunkt (-1, -1, -1) og (1, 1, 1). Dette gjøres for å effektivisere klippingen i pipelinen.

Det finnes hovedsaklig to typer projeksjoner; ortografisk (eller parallell) og perspektiv projeksjon. Hovedkarakteristikken til en ortografisk projeksjon er at parallelle linjer skal forbli parallelle etter transformen. Perspektiv projeksjoner er litt mer kompliserte, men hovedtrekket er at objekter lenger unna skal være mindre enn objekter som nærme kameraet. Denne typen projeksjoner prøver altså å imitere hvordan det menneskelige øyet ser verden. Synsvolumet til en perspektiv projeksjon er formet som en avkortet pyramide som på figur 2.3.

Kun de grafiske primitivene som er fullstendig eller delvis innenfor synsvolumet skal tegnes til skjermen. Et primitiv som er delvis innenfor synsvo-



Figur 2.4: *Primitiver som ligger delvis innenfor enhetskuben må klippes slik som vist på figuren. Etter klipping blir punktene som ligger utenfor enhetskuben erstattet med nye punkt der hvor den skjærer enhetskuben.*

lumet må klippes før det sendes videre nedover i pipelinen. Fordelen ved å foreta klipping etter projeksjonsfasen i geometripipelinen er at alle primitiver klippes mot samme enhetskube og klippingen kan optimaliseres deretter. Denne prosessen er vist på figur 2.4.

Alle grafiske primitiver som er synlig innenfor synsvolumet blir sendt videre til siste delen i geometripipelinen : Skjermavbildning. Her blir (x, y) koordinatene til alle synlige primitiver i enhetskuben gjort om til skjermkoordinater. z-koordinaten blir ikke påvirket av skjermavbildningen. Avbildningen finner ut hvor på skjermen et synlig primitiv skal tegnes og transformerer til det lokale skjermsystemet.

2.1.3 Rasteriseringsfasen

Grafikksystemer er som oftest rasterbasert - et digitalt bilde på en skjerm består av et raster av små bildeelementer kalt piksler. Akkurat som geometrifasen bestod av operasjoner for å håndtere grafiske primitiver, består rasteriseringsfasen av operasjoner for å gjøre om disse primitivene til piksler. Også denne fasen foregår på grafikkakseleratoren.

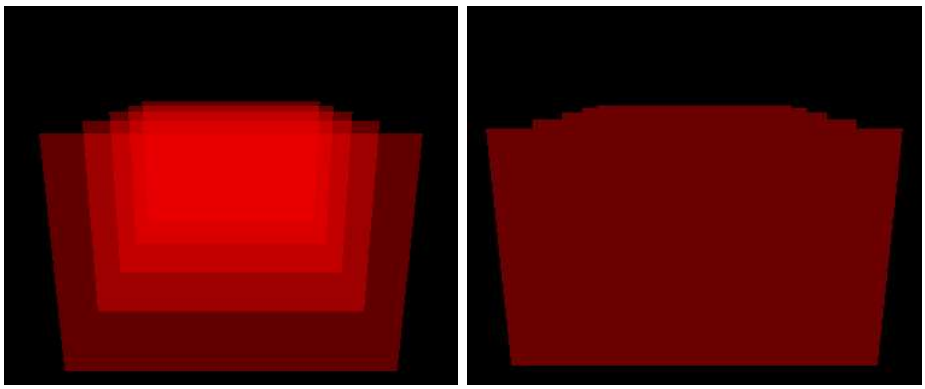
Vi beregner en tilhørende fargeverdi til alle piksler som et grafisk primitiv består av. Denne fargeverdien består av tre komponenter (rødt, grønt og blått), og skrives ofte som RGB. Fargeverdiene til alle tilhørende piksler på skjermen blir lagret i et spesielt fargebuffer.

På samme måte som vi beregner fargen for en piksel og lagrer resultatet i et fargebuffer kan vi assosiere z-verdien til alle piksler i et eget dybdebuffer. Siden z-verdien angir hvor langt unna en piksel er i forhold til det virtuelle kameraet kan vi bruke denne til å undersøke om en piksel er synlig. Synligheten til en piksel må undersøkes fordi et grafisk primitiv innenfor synsvolumet vårt kan fortsatt være helt eller delvis skjult bak et annet primitiv. Andre operasjoner som foregår under rasteriseringsfasen er teksturering og alfablending.

Innenfor datagrafikk bruker vi definisjonen tekstur på en bilderepresentasjon av et objekt. Har vi to objekter med lik størrelse og geometri, er det ved hjelp av teksturen til objektene at vi klarer å skille dem fra hverandre. Teksturering er altså en prosess som tar et grafisk primitiv og forandrer utseendet ved å lime et digitalt bilde på primitiven.

Alfablending lar oss forbinde en alfaverdi til en piksel i tillegg til fargeverdiene. Alfaverdien, ofte skrevet som α, beskriver en flates opasitet som er et mål på hvor mye lys som penetrerer gjennom flaten. En alfaverdi på 1 korresponderer til en helt opak flate: ikke noe lys slipper gjennom den. Er $\alpha = 0$ har vi en flate som er helt gjennomsiktig fordi alt lys slipper gjennom den. Alfaverdier mellom 0 og 1 angir ulike grader av gjennomskinnelighet.

Alfablending er avhengig av rekkefølgen de grafiske primitivene blir sendt til prosessering i grafikkakseleratoren. For å oppnå riktig visuell resultat må vi først tegne alle opake primitiver, for så å tegne de gjennomskinnelige primitivene bak til front. Dette må gjøres fordi blendingsfunksjonen som beregner fargen til et primitiv tar hensyn til fargeverdien som allerede finnes for pikselen. Figur 2.5 viser hvor viktig det er å blende primitiver i riktig rekkefølge.



Figur 2.5: *Eksempel på hvor viktig det er å tegne gjennomskinnelige primitiver i riktig rekkefølge. Til venstre ser vi 5 plan med $\alpha = 0.4$ som er tegnet bak til front. Til høyre ser vi resultatet av å tegne de samme planene med samme alfaverdi front til bak. Vi ser at resultatet til høyre ikke gir riktig visuelt resultat.*

2.2 OpenGL

I denne oppgaven bruker vi OpenGL som verktøy for å oppnå sanntidsvisualisering. OpenGL er et API ("application programming interface") rettet mot datagrafikk. Det er laget for å være uavhengig av spesifikk maskinvare og operativsystem, og inneholder derfor ikke operasjoner for å håndtere spe-

sielle funksjoner i et grafisk operativsystem. Derimot har OpenGL en rekke funksjoner for å behandle grafiske primitiver.

OpenGL er konstruert som en tilstandsmaskin; tilstander som blir satt i OpenGL varer helt til man endrer tilstanden. Et eksempel på en slik tilstand i OpenGL er opptegningsfarge. Setter man for eksempel opptegningsfargen til blå ved et OpenGL kall, vil alle grafiske primitiver som tegnes til skjermen være blå helt til man gir OpenGL en ny opptegningsfarge. Eksempler på andre tilstander som blir satt i OpenGL er linje- og punktstørrelse, samt lysegenskaper.

2.3 Objektorientering

I objektorientering arbeider man med abstrakte objekter og relasjonene mellom disse. En klasse søker å beskrive egenskapene til et fenomen som et dataprogram må forholde seg til. Disse egenskapene er representert ved variable og funksjoner.

Har vi først beskrevet en klasse kan vi lage et objekt av denne. Et objekt har egen tilstand (data) og oppførsel (funksjoner). Objekter består altså av sine egne lokale data samt funksjoner som aksesserer eller endrer disse. Alle objekter av samme klasse har felles operasjoner men (generelt) ulik tilstand. Vi kaller prosessen med å lage et objekt fra en klasse for instansiering, og det ferdige objektet for en instans.

En klasse kan også arve egenskaper fra en annen klasse. Arv i objektorientering fører til at vi får en hierarkisk struktur. Klasser nederst arver operasjoner fra klassene over dem i hierarkiet.

Ofte er det slik at problemet vi ønsker å modellere har flere like komponenter som det kan være lurt å dele inn i en hierarkisk struktur. Fordelene ved å gjøre dette i et dataprogram er blant annet:

- *Relasjon:* Ved å bruke relasjoner mellom klasser som gjenspeiler det virkelige fenomenet oppnår man en abstraksjon som gjør det enkelt å utvide programmet ved en senere anledning. Bruken av klasser for å beskrive et fenomen hjelper oss også med å minske avstanden mellom konsept og implementasjon.

- *Gjenbruk av kode:* Lager man en klassebeskrivelse av et fenomen for å løse et gitt problem, kan man enkelt bruke dette i andre program for å løse lignende problemer uten å skrive alt på nytt. Arv gjør det også lett å utvide et allerede eksisterende program.

Kapittel 3

Modeller

Modeller er abstraksjoner - både av den virkelige verden og av vår virtuelle verden implementert i en datamaskin. I mange år har mennesket brukt matematiske modeller for å simulere fenomener som oppstår i naturen. Eksempler på slike modeller er partielle differensialligninger for å beskrive varmeledning eller statistiske modeller for å bedre forstå vær- og vindforhold. Felles for alle disse modellene er at de bruker ligningssett for å approksimere det fysiske fenomenet man ønsker å studere. Hva slags ligninger, eller nærmere bestemt hva slags matematikk man velger å bruke, avhenger av hvordan fenomenet oppfører seg og av egenskapene til det matematiske ligningssettet.

Selv om grafikkakseleratorer stadig øker prosesseringshastiget, øker kompleksiteten i våre tredimensjonale modeller tilsvarende, som regel målt etter hvor mange grafiske primitiver den består av. Fortsatt er antall primitiver vi ønsker å tegne til skjermen langt større enn antallet primitiver grafikkakseleratorer klarer å prosessere i sanntid. For å beholde vårt krav om sanntidsvisualisering må vi hele tiden sørge for å minimere antall grafiske primitiver som blir behandlet av grafikkakseleratoren. Dette var som vi nevnte i seksjon 2.1 en av hovedoppgavene til applikasjonsfasen. Innenfor datagrafikk bruker en applikasjon ulike akselerasjonsalgoritmer som har som mål å hjelpe oss med å effektivisere visualiseringsprosessen.

Vi skal i dette kapittelet se på to slike akselerasjonsalgoritmer. Den første akselerasjonsalgoritmen hjelper oss med å ordne geometri i rommet. Siden vi i denne oppgaven arbeider med skog ønsker vi å visualisere store terrengområder. For å effektivisere denne prosessen søker vi datastrukturer som gjør det lettere å håndtere store datasett. Slike datastrukturer kalles romlige datastrukturer [9, side 246]. I denne oppgaven skal vi se på en type romlig datastruktur kalt et quadtre.

Vi skal se at vi effektivt kan luke bort store mengder geometrisk informasjon fra terrenget ved å bruke et quadtre. Men selv om quadtreet fjerner en del unødvendig informasjon, har vi fortsatt (i vårt tilfelle) mange tusen trær som må tegnes til skjerm. “Level-of-Detail” er en metode som hjelper

til med å minske den geometriske kompleksiteten til et objekt avhengig av hvor stort bidrag objektet gir til det totale bildet [9, side 289].

3.1 Romlige datastrukturer

Romlige datastrukturer ordner geometri i et n-dimensjonalt rom. Slike datastrukturer kan blant annet brukes til spørringer om overlappende geometri. Et annet bruksområde er å undersøke om et objekt er innenfor synsvolumet til det virtuelle kameraet. Romlige datastrukturer er ofte ordnet i et hierarki; den øverste noden i hierarkiet spenner over det romlige området til nivået under i hierarkiet. Dette nivået spenner igjen over området til sine egne barn og så videre nedover i hierarkiet. Slike datastrukturer er nesten alltid nøstet og rekursive av natur.

3.1.1 Quadtre

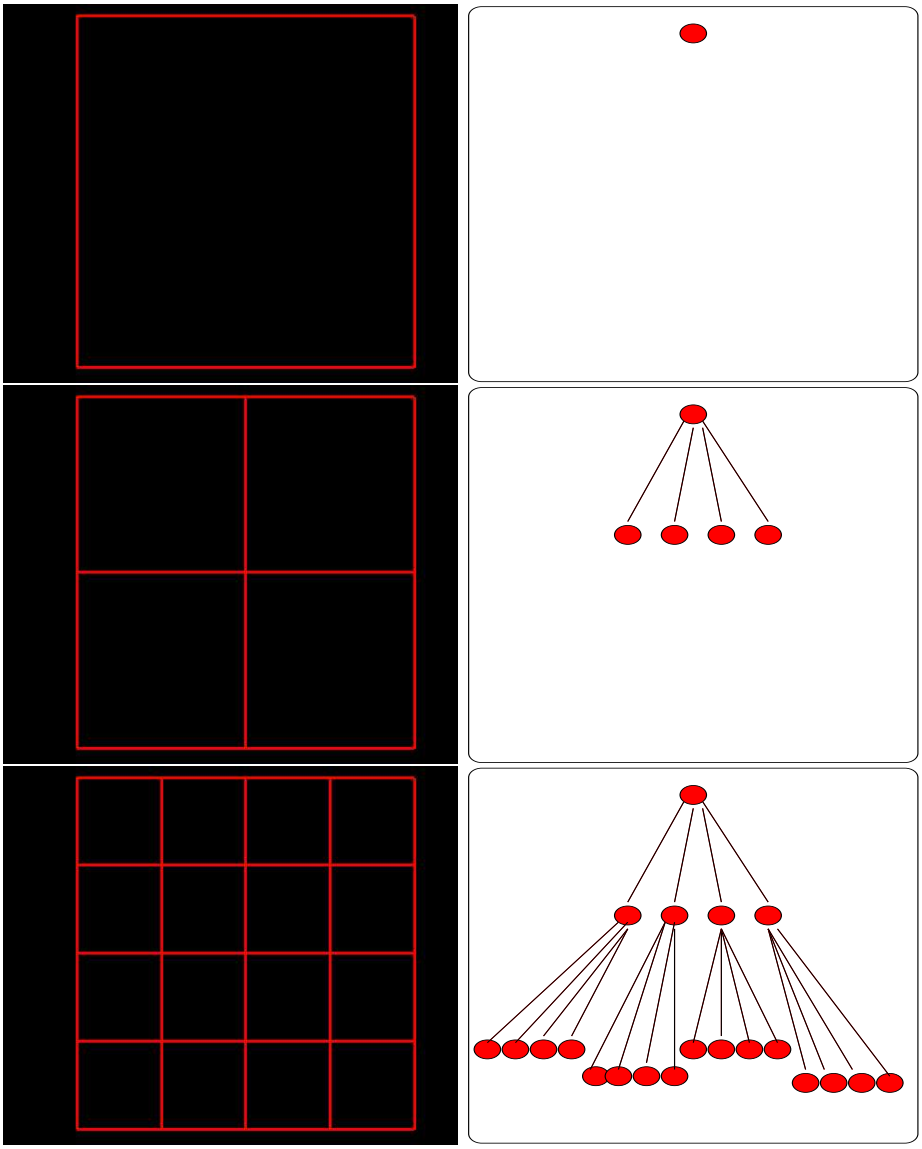
Et quadtre er en rekursiv trestruktur som gjør det lett å traversere en hierarkisk modell over et planart område. Quadtreet lages ved å innkapsle et rektangulært område i rotnoden, for så å splitte halvveis i begge retninger for å få fire kvadranter som hver dekker et like stort område. Disse fire kvadrantene, som nå er rotnodens barn, blir så rekursivt delt opp i fire nye kvadranter hver. Denne prosessen er vist på figur 3.1.

Et annet ord for en node i quadtreet er en tile. Hver tile i quadtreet sies å være på et bestemt nivå, hvor vi sier at rotnoden av konvensjon befinner seg på nivå 0. Barna til rotnoden finnes dermed på nivå 1 og så videre nedover i treet. Dybden til quadtreet er dermed tallet som tilsvarer hvor mange nivåer treet består av. En gitt dybde n betyr dermed at det laveste nivået til et barn må være $n - 1$ (på grunn av rotnoden på nivå 0). For eksempel har quadtreet nederst på figur 3.1 en dybde på 3, men nivået til det dypeste barnet er 2.

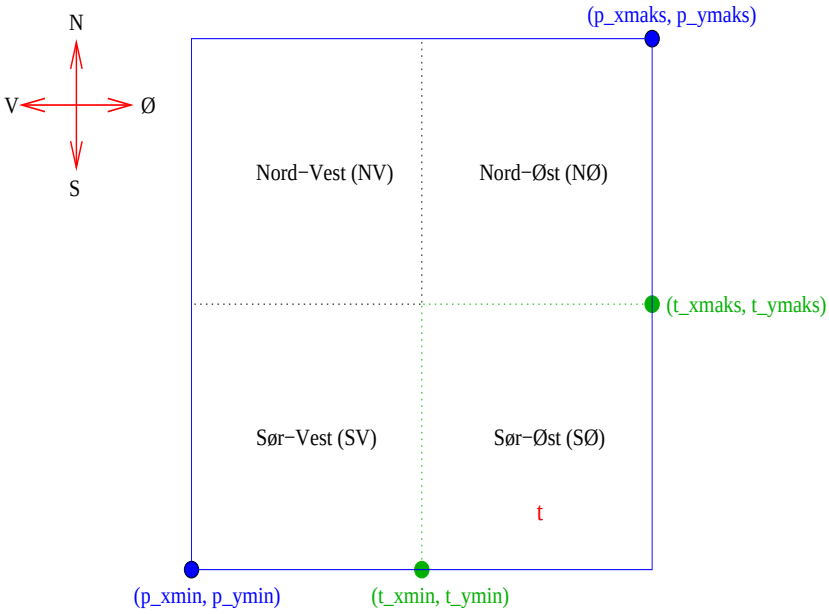
Quadtreet er regulært i den forstand at det inndeler et plan etter et uniformt mønster uten overlappende geometri. Denne inndelingen foregår rekursivt nedover i hierarkiet til et gitt stoppkriterium er oppfylt. Eksempler på stoppkriterium kan være maksimal dybde på quadtreet, maksimalt antall grafiske primitiver i en tile eller lignende.

Vi forbinder hver tile i quadtreet med sin egen orientering i forhold til foreldrenoden. Denne orienteringen er basert på vanlig kompassretning som på figur 3.2. Ved å utnytte orienteringen til en tile t i forhold til sine foreldre kan man lett beregne grensene til t ut i fra foreldretilens grenser.

La oss nå anta at vi ønsker å finne grensene $(t_{x_{min}}, t_{y_{min}})$ og $(t_{x_{maks}}, t_{y_{maks}})$ til en tile t. Vi vet kun grensene til foreldenoden p gitt som $(p_{x_{min}}, p_{y_{min}})$ og $(p_{x_{maks}}, p_{y_{maks}})$ som på figur 3.2. Vi begynner med å finne x_Δ og y_Δ gitt som:



Figur 3.1: *Et eksempel på et quadtre. Øverste rad viser rotnoden i quadtreet som dekker hele planet vårt. Midterste rad viser neste nivå i quadtreet, hvor vi har delt inn rotnoden i fire like store kvadranter. Nederste rad viser alle barna på nivå 2 i quadtreet ved å splitte rotnodens fire barn. Venstre kolonne viser en grafisk representasjon av quadtreet; høyre kolonne viser en tegning av objektrelasjonene.*



Figur 3.2: *Vi forbinder ethvert barn til en node i quadtreet med en unik orientering i forhold til foreldrenoden. Denne orienteringen blir forbundet med et heltall som hjelper oss med å beregne de riktige grensene til en node avhengig av grensene til foreldrenoden.*

$$x_\Delta = \frac{1}{2}(p_{x_{maks}} - p_{x_{min}}), \tag{3.1}$$

og

$$y_\Delta = \frac{1}{2}(p_{y_{maks}} - p_{y_{min}}). \tag{3.2}$$

Både x_Δ og y_Δ angir da halve avstanden mellom p sine grenser. Har vi disse, ser vi at for tilen t på figur 3.2 er grensene til t gitt som:

$$\begin{aligned} t_{x_{min}} &= p_{x_{min}} + x_\Delta && (3.3)\\ t_{y_{min}} &= p_{y_{min}} && (3.4)\\ t_{x_{maks}} &= p_{x_{maks}} && (3.5)\\ t_{y_{maks}} &= p_{y_{min}} + y_\Delta && (3.6) \end{aligned}$$

Gitt foreldrenoden p og orienteringen o_t til en tile t, finner vi grensene til t ved følgende algoritme:

Algoritme 3.1, beregnGrenser(p, o_t)

```
1
2   x_Δ = ½ (p_xmaks − p_xmin)
3   y_Δ = ½ (p_ymaks − p_ymin)
4   hvis o_t tilhører den nordlige halvdelen til p
5       t_ymin = p_ymin + y_Δ
6       t_ymaks = p_ymaks
7   ellers
8       t_ymin = p_ymin
9       t_ymaks = p_ymin + y_Δ
10  hvis o_t tilhører den østlige halvdelen til p
11      t_xmin = p_xmin + x_Δ
12      t_xmaks = p_xmaks
13  ellers
14      t_xmin = p_xmin
15      t_xmaks = p_xmin + x_Δ
```

3.1.2 Hierarkisk synlighetsspørring

Vi skal nå se på hvordan vi kan utnytte quadtrestrukturen for å minimere antall grafiske primitiver som må sendes videre til prosessering i geometrifasen. For å gjøre dette må vi sjekke om et objekt er synlig innenfor synsvolumet til det virtuelle kamera. En slik synlighetsspørring må være rask siden den alltid foregår under kjøringen av applikasjonen.

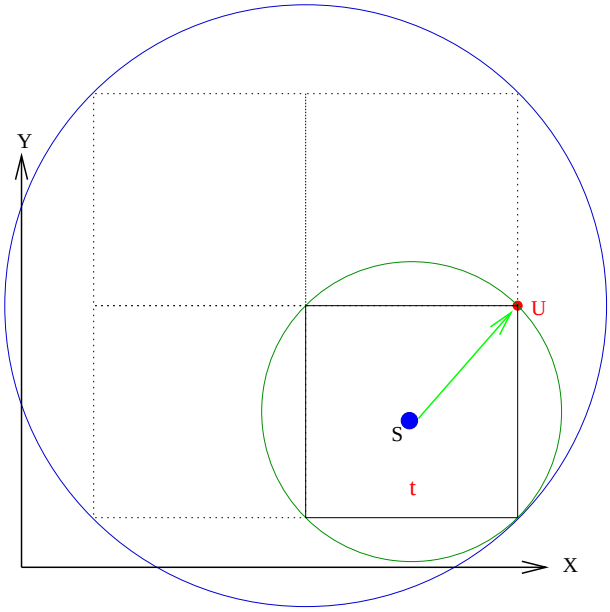
Vi ser av figur 3.1 at de fire barna til en tile t i quadtreet tilsammen utgjør det totale området til t. En vilkårlig tile i quadtreet er altså fullstendig inneholdt i sin foreldretile. Vi har da at quadtreet er en hierarkisk romlig datastruktur, og vi kan utnytte egenskapene til hierarkiet for å optimalisere synlighetsspørringen.

Hvis det nå skulle vise seg at t ligger utenfor synsvolumet, så vet vi at også barna til t ligger utenfor synsvolumet (siden barna til t spenner ut samme område som t). Vi oppnår dermed en hierarkisk synlighetsspørring. Denne spørringen må foregå for hver tile t under traverseringen av quadtreet.

For å effektivt undersøke om en tile t er innenfor synsvolumet begynner vi med å assosiere en omringet sfære til t. Sfæren har senter S i midtpunktet til t med radius:

$$r = \| U - S \|, \tag{3.7}$$

hvor $U = (t_{x_{maks}}, t_{y_{maks}})$ som på figur 3.3 og $\| \cdot \|$ er vanlig euklidsk avstand. Vi ser på figur 3.3 at akkurat som t er inneholdt i foreldretilen p, så er den omringede sfæren til t inneholdt i den omringede sfæren til p. Grunnen til at vi ønsker å assosiere en omringet sfære til en tile i quadtreet vårt er at synlighetsspørringer mot en sfære er meget effektive.



Figur 3.3: *Figuren viser hvordan vi assosierer en omringet sfære til en tile t. Gitt midtpunktet S og punktet* $U = (t_{x_{maks}}, t_{y_{maks}})$*, er radien til sfæren gitt ved* $|U - S|$*. Vi ser også at den omringede sfæren til foreldretilen inneholder den omringede sfæren til t.*

Begynn med å finne ligningen for et plan ved

$$Ax + By + Cz + D = 0, \tag{3.8}$$

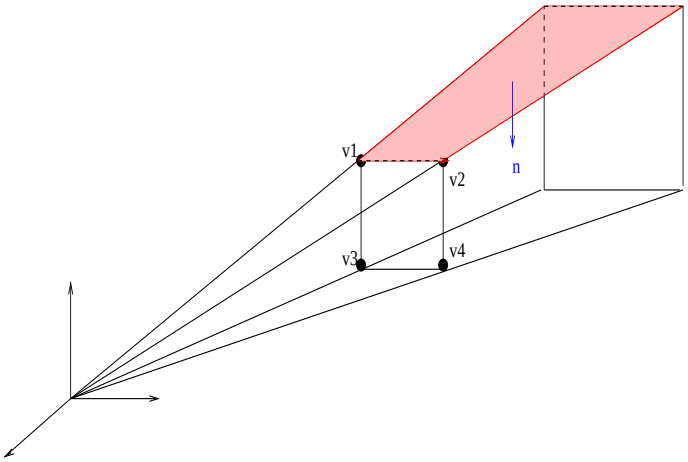
som kan også skrives på formen

$$\mathbf{n} \cdot \mathbf{p} + d = 0, \tag{3.9}$$

hvor $\mathbf{n}$ er normalen til planet, $\mathbf{p}$ er et tilfeldig punkt i planet og d er en konstant som forteller oss hvor planet ligger i rommet. Når vi bruker en perspektivtransform for å sette opp vårt synsvolum, vet vi at fire av de seks planene som utgjør synsvolumet går gjennom origo som på figur 3.4. Dermed blir d i ligning (3.9) 0 for disse fire planene. Med $d = 0$ i ligning (3.9), er planet kun gitt av normalen $\mathbf{n}$.

La oss nå anta at vi ønsker å finne ligningen for topplanet i synsvolumet vårt. Siden $d = 0$ i ligning (3.9) må vi kun finne normalen $\mathbf{n}$ til dette planet. Vi ønsker at $\mathbf{n}$ skal peke innover i synsvolumet som på figur 3.4. Gitt de to punktene v_1 og v_2 for synsvolumet som på figur 3.4, kan vi finne $\mathbf{n}$ ved å ta kryssproduktet:

$$\mathbf{n} = \| \vec{u_1} \times \vec{u_2} \|, \tag{3.10}$$



Figur 3.4: *Figuren viser hvordan vi kan utnytte egenskapene til synsvolumet for å finne de ulike ligningene til de seks planene som synsvolumet består av.*

hvor $\vec{u_1}$ er den normaliserte vektoren fra origo til punktet v_1 og $\vec{u_2}$ er den normaliserte vektoren fra origo til punktet v_2. Rekkefølgen på kryssproduktet i ligning (3.10) sørger for at $\mathbf{n}$ peker inn mot synsvolumet. Tilsvarende kan vi finne normalene til de tre resterende planene ved å ta ulike kryssprodukt av v_1, v_2, v_3 og v_4 som i figur 3.4. De to resterende planene i synsvolumet på figur 3.4 er klippeplanene som vi spesifiserer når vi setter opp synsvolumet i OpenGL.

Har vi funnet ligningen for et plan i synsvolumet gjenstår det kun å sjekke om et punkt $\mathbf{p}$ er synlig innenfor synsvolumet. Dette gjøres enkelt ved å sjekke mot radien til den omringede sfæren. La nå s være senteret til sfæren B som tilhørerer en tile t, og la r være radien til B. Vi ønsker å sjekke om B ligger fullstendig utenfor synsvolumet vårt. Denne prosessen vises for et plan i synsvolumet vårt gitt som $\mathbf{n} \cdot \mathbf{p} = d$.

En sfære B ligger fullstendig utenfor synsvolumet hvis [10, side 158]:

$$\mathbf{n} \cdot \mathbf{s} - d < -r, \tag{3.11}$$

altså hvis avstanden fra planet d i synsvolumet til senteret s i sfæren B er større enn radien r til B. Høyre siden i ligning (3.11) er negativ fordi vi ønsker å finne ut om B ligger utenfor synsvolumet (husk at $\mathbf{n}$ peker innover i synsvolumet). Er ligning (3.11) oppfylt kan vi la være å sende tilen t (og hele subtreet til t) ned til videre prosessering i geometrifasen.

3.2 "Level-of-Detail" (LOD)

I datagrafikk er det alltid en avveining mellom kompleksiteten i en scene og ytelsen. Eller sagt på annen måte; mellom hastighet og realisme. Innenfor datagrafikk har det vokst et eget felt ut av denne problemstillingen som har som mål å bygge en bro mellom kompleksitet og ytelse ved å regulere detaljnivået til objektene i en scene. Dette feltet kalles "Level-of-Detail" modellering, forkortet LOD.

Den mest åpenbare løsningen er å organisere våre objekter i forhold til avstanden til kameraet. Mer generelt kan vi si at vi ønsker å organisere objektene etter hvor stort bidrag de gir til det totale bildet. Sett nå at vi har et tredimensjonalt objekt bestående av flere tusen grafiske primitiver. Jo lenger unna dette objektet befinner seg fra det virtuelle kameraet, jo mindre blir dette objektet på skjermen. Hvis vi utnytter denne relasjonen, kan vi approksimere et geometrisk komplisert objekt ved hjelp av færre grafiske primitiver når dette objektet er langt unna, det vil si mindre på skjermen. Dette er den fundamentale tankegangen bak LOD.

For å få til dette må vi ha flere detaljrepresentasjoner av våre tredimensjonale modeller. Helst bør det være slik at antallet grafiske primitiver fordobles fra en representasjon til en annen. Vi får dermed et hierarki av detaljrepresentasjoner; for hvert nivå høyere opp i hierarkiet får vi finere og finere representasjoner av vårt originale objekt.

Etter at vi har ulike representasjoner av en modell må vi ha en mekanisme for å bestemme når vi skal skifte mellom de ulike LOD representasjonene. Denne seleksjonsmekanismen bestemmer til enhver tid hvilken modell som skal velges ut i fra et sett med gitte kriterier. Til slutt må vi foreta selve skiftet fra en "Level-of-Detail" representasjon til en annen.

3.2.1 Generering, seleksjon og skift

Det finnes hovedsaklig to ulike fremgangsmåter for å bygge opp et "Level-of-Detail" hierarki. Disse er kalt Diskret og Kontinuerlig LOD [11, side 9] [1]:

- *Diskret LOD:* Dette er den tradisjonelle måten å håndtere et LOD hierarki på. Man genererer en mengde ulike detaljerte representasjoner av et objekt som en preprosess, organiserer disse i et hierarki, og velger passende LOD representasjon under kjøringen av applikasjonen. Siden "Level-of-Detail" representasjonene genereres før programmet kjøres, er det ikke mulig å vite fra hvilket synspunkt objektet vil bli sett fra. Dermed er det vanlig under preprosesseringen å forenkle uniformt over hele modellen.

[1] Merk at begrepene skiller seg fra det vi mener med diskrete og kontinuerlige mengder innenfor matematikken

- *Kontinuerlig LOD:* I stedet for å forhåndsgenerere en mengde "Level-of-Detail" modeller, kan man velge å konstruere LOD hierarkiet slik at man oppnår et kontinuerlig spektrum av ulik detaljnivå. Ønsket detaljnivå for en modell blir beregnet for hver bilderamme istedet for å velge mellom et gitt antall modeller. Fordelen ved dette er at man til enhver tid oppnår full kontroll over mengden grafiske primitiver som tegnes til skjerm.

Synsavhengig LOD viderefører tankegangen bak kontinuerlig LOD, men bruker synsavhengig kriterium for å velge den mest egnede detaljgraden for en gitt synsvinkel. Et enkelt objekt kan bestå av flere ulike nivåer til enhver tid, avhengig av hvor det virtuelle kamera befinner seg i forhold til objektet. Store datasett bruker ofte denne teknikken for å skille ut deler av datasettet som er nærme kameraet fra de delene som befinner seg lengre unna. Subsettet av det originale datasettet som befinner seg nærme kameraet kan dermed tegnes med større detaljgrad enn de delene av datasettet som er lengre unna.

Quadtreet egner seg godt til denne typen synsavhengig LOD. Vi bruker nivåene i quadtreet som om de er ulike nivåer i et "Level-of-Detail" hierarki: rotnoden på nivå 0 har dårligst oppløsning, og vi får finere og finere oppløsning jo lenger ned i quadtreet vi traverserer. Dette tilsvarer finere og finere inndeling av planet som quadtreet innkapsler. De individuelle trærne i terrenget representeres ved ulike diskret LOD modeller. Dette vil si at alle trerepresentasjonene for skogmodellen blir generert under preprosessering.

Gitt flere diskret LOD representasjoner er neste skritt å velge riktig representasjon ut i fra et sett med kriterier. Noen eksempler på slike kriterier kan være geometrisk avstand, antall piksler objektet utspenner eller det totale arealet av objektet på skjermen. Det er seleksjonsmekanismen til vår "Level-of-Detail" implementasjon som vil bestemme når vi skal tegne en gitt detaljgrad.

La oss nå anta at vi ønsker å bruke et avstandsmål avhengig av synspunktet og lokasjonen til et gitt objekt. Vi har tre ulike "Level-of-Detail" representasjoner av et objekt hvor LOD_2 er den fineste representasjonen. Seleksjonsmekanismen velger LOD_2 hvis:

$$0 \leq r < r_1. \tag{3.12}$$

Her er r avstanden fra kameraet til LOD_2 representasjonen, og r_1 er et tall som angir når denne avstanden blir for stor. På samme måte skal LOD_1, den neste representasjonen i "Level-of-Detail" hierarkiet, velges når:

$$r_1 \leq r < r_2, \tag{3.13}$$

hvor $r_1 < r_2$. LOD_0 representasjonen velges hvis avstanden r tilfredstiller:

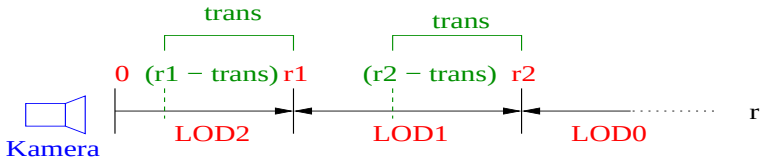
$$r \geq r_2. \tag{3.14}$$

Denne prosessen er illustrert på figur 3.5.

Figur 3.5: *Eksempel på når man skal velge tre ulike "Level-of-Detail" representasjoner avhengig av gitte avstandsmål r_1 og r_2.*

Når vi skifter fra en "Level-of-Detail" representasjon til en annen, kan det forekomme brå synlige hopp som i LOD teorien kalles for popping. Slike effekter er ofte en distraksjon for brukerne av en applikasjon og vi ønsker derfor å unngå dette så langt det lar seg gjøre. I denne oppgaven bruker vi en variant av "Blend LODs" for å forhindre popping [9, side 391].

For å glatte over hvert skift mellom de ulike representasjonene i et LOD hierarki kan vi bruke alfablending. Sett nå at vi ønsker å skifte LOD representasjon fra LOD_1 til LOD_2 for et objekt. Dette vil si at kameraet er plassert i avstand $r = r_1$ og at vi beveger objektet mot kameraet på figur 3.5.



Figur 3.6: *Eksempel på hvordan vi knytter blendinglengden trans og LOD_i skiftene r_i til et avstandsmål r.*

Når $r = r_1$ på figur 3.6 tegnes LOD_1 representasjonen som opak. Etterhvert som avstanden r blir mindre ønsker vi å blende inn LOD_2 samtidig som vi blender ut LOD_1. For å hjelpe oss med dette innfører vi en blendinglengde som bestemmer over hvor lang avstand vi skal blende inn og ut de ulike "Level-of-Detail" representasjonene. Vi kaller blendinglengden for *trans*.

Når avstanden $r = r_1$ på figur 3.6 skal LOD_1 representasjonen ha $\alpha = 1$ og LOD_2 ha $\alpha = 0$. Etterhvert som avstanden minker til $(r_1 - trans)$ skal LOD_1 få stadig mindre alfaverdi, samtidig som LOD_2 sin alfaverdi økes. Når $r = (r_1 - trans)$ skal alfaverdien til LOD_1 være 0, samtidig som alfaverdien til LOD_2 er 1. Ulempen med denne fremgangsmåten er at vi under blendingsprosessen må tegne to modeller oppå hverandre.

Kapittel 4

Oppbygning av skogmodell

Å modellere et så komplisert naturlig fenomen som en skog nøyaktig er en nesten umulig oppgave for en datamaskin. Vi søker derfor en modell som på best mulig måte approksimerer en skog. Målet er å oppnå en fleksibel hierarkisk modell som genererer en naturtro representasjon.

Vi begynner dette kapittelet ved å introdusere datagrunnlaget vårt. For å vite noe om landskapet må vi ha et datasett som innfører informasjon om terrenget. I vårt tilfellet er denne informasjonen organisert i et raster.

Etter at vi har introdusert datasettet ser vi på hvordan vi kan bruke quadtreet til å inndele landskapet i et hierarki. Vi ønsker å utplassere treposisjoner i quadtreet slik at flere trær blir synlige jo nærmere man befinner seg landskapet. For å lage en mer realistisk skogmodell bør vi også være i stand til å utplassere en rekke ulike tretyper. Til slutt ser vi på hvordan skogmodellen kan genereres dynamisk under kjøringen av applikasjonen.

4.1 Datagrunnlag

Som nevnt i innledningen må vi begrense vår skogmodell til å best mulig approksimere en skog. Vi skal her se på hva slags datagrunnlag vi jobber med i denne oppgaven. Datagrunnlaget gjør det mulig å knytte informasjon til landskapet slik at man kan finne ut hva slags terreng man arbeider med.

4.1.1 Rasterbasert landskapsinformasjon

Med antagelsene i seksjon 1.3.1 er vi klare for å takle spørsmålet: Hvordan lage en skogmodell? Problemstillingen er midlertidig noe forenklet fordi vi sier at en skog i vårt tilfelle kun er en samling av trær. Dermed blir løsningen å utplassere trær i et gitt landskap for å skape vår virtuelle skog. Den geometriske posisjonen til det tre i terrenget kaller vi for en treposisjon.

For at vi i det hele tatt skal kunne utplassere treposisjoner i landskapet må vi vite noe om terrenget. Har vi for eksempel åpent vann, tettbebyggelse

eller andre typer terreng der trær ikke vokser, må vi sørge for å ikke utplassere treposisjoner i slike områder. Vi må altså ha muligheten til å undersøke terrengtype.

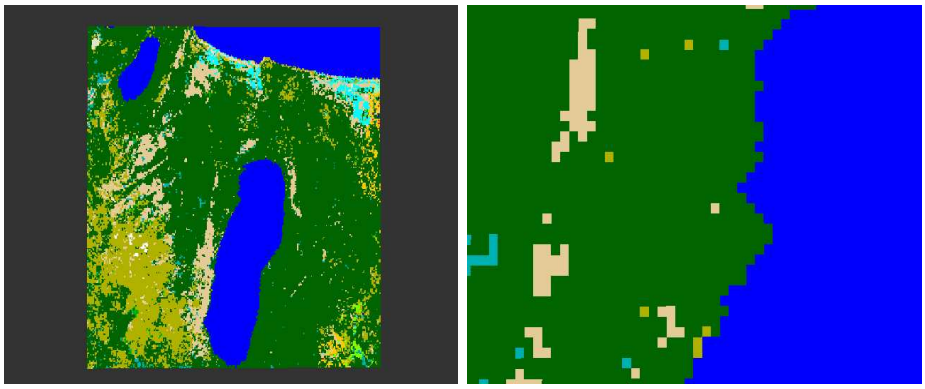
Datagrunnlaget vi har arbeidet med i denne oppgaven er gitt som et raster. Et raster er et rutemønstret regulært grid som består av kvadratiske elementer. Vi kaller et element i rasteret for en celle. En celle i rasteret spenner over et gitt område i terrenget og det er oppløsningen på cellen som bestemmer hvor stort dette området er. Hver celle i rasteret er forbundet med en terrengidentifikasjon som angir hva slags terreng cellen representerer.

I denne oppgaven arbeider vi med et rasterbasert datasett utarbeidet av "The U.S. Geological Survey (USGS)" i samarbeid med "U.S. Environmental Protection Agency". Dette datasettet kalles "The National Land Cover Data", forkortet NLCD. Oppløsningen på en celle i rasteret er 30×30 meter og inneholder 21 kategorier for terrengidentifikasjon. Inndelingen av terrengidentifikasjon for en celle er gitt ut i fra informasjon fra satellittbilder.

De 21 terrengkategoriene for NLCD datasettet er forbundet med et heltall som gir et unikt identifikasjonsnummer. Spesifikasjonen på NLCD dataformatet er gitt som:

```
NLCD Land Cover Classification System Key - Rev. July 20, 1999

Water
 11 Open Water
 12 Perennial Ice/Snow

Developed
 21 Low Intensity Residential
 22 High Intensity Residential
 23 Commercial/Industrial/Transportation

Barren
 31 Bare Rock/Sand/Clay
 32 Quarries/Strip Mines/Gravel Pits
 33 Transitional

Forested Upland
 41 Deciduous Forest
 42 Evergreen Forest
 43 Mixed Forest

Shrubland
 51 Shrubland

Non-natural Woody
```

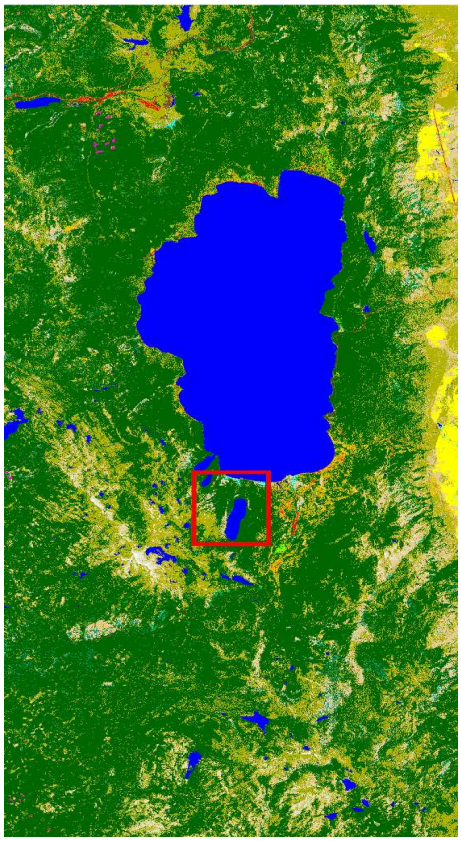
```
 61 Orchards/Vineyards/Other

Herbaceous Upland
 71 Grasslands/Herbaceous

Herbaceous Planted/Cultivated
 81 Pasture/Hay
 82 Row Crops
 83 Small Grains
 84 Fallow
 85 Urban/Recreational Grasses

Wetlands
 91 Woody Wetlands
 92 Emergent Herbaceous Wetlands
```



Figur 4.1: *NLCD datasettet som vi arbeider med i denne oppgaven hentet fra området rundt "Lake Tahoe" i USA. Landskapet er gitt som et* 256×256 *raster (*65536 *celler), hvor hver celle har en oppløsning på* 30×30 *meter. På figurene er hver celle i rasteret fargelagt avhengig av terrengidentifikasjon. Ved å gå nærmere inn på landskapet ser vi på bildet til høyre de individuelle cellene i rasteret.*

Figur 4.1 viser NLCD datasettet som vi har arbeidet med i denne oppgaven. Området består av et 256×256 raster rundt "Lake Tahoe" regionen i Amerika, og er et lite utsnitt av hele regionen vist på figur 4.2. Et lite utsnitt av det tilhørende rasteret til figur 4.1 ser ut som følger:

42	42	42	...	11	11	11
42	42	42	...	11	11	11
⋮	⋮	⋮	⋮	⋮	⋮	⋮
42	41	41	...	42	42	42
43	51	51	...	42	42	42



Figur 4.2: *Her ser vi hele "Lake Tahoe" regionen. Datasettet vi arbeider med er markert med rødt.*

Vi ser at øverste hjørnet til venstre er av type skog (42), mens hjørnet øverst til høyre er vann (11). Tilsvarende kan vi se at de nederste hjørnene i rasteret tilsvarer landskapet gitt på figur 4.1. Gitt en klassifikasjon av terrenget som NLCD datasettet og et tilhørende raster er dette all informasjonen vi behøver for å bygge opp skogmodellen. Ved å bruke denne klassifikasjonen vet vi at celler med identifikasjonsnummer 41, 42 og 43 er av type "skog" og det er dermed kun de vi interesserer oss for.

Merk at vi allerede her må gjøre noen valg som kan gi urealistisk gjengivelse. Ved å ignorere alle andre terrengtyper i vårt datasett, vil heller ingen treposisjoner bli utplassert i disse. For å få en helt realistisk modell må treposisjoner utplasseres enkeltvis. Dette er midlertidig ikke en aktuell løsning, da et av målene med skogmodellen er at den skal genereres under oppstart og kjøring av applikasjon. Vi må dermed foreta avskjæringer i vår modell og ignorere potensielle trær i andre terrengtyper.

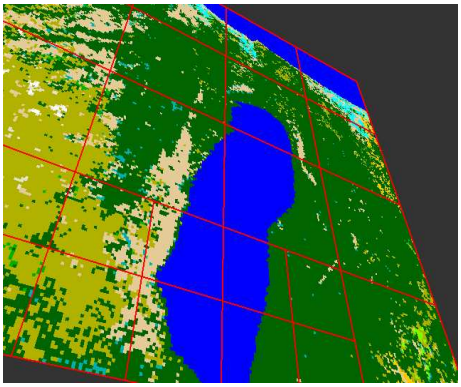
NLCD datasettet skiller mellom to typer skog: løvskog ("Deciduous", terrengidentifikasjon 41) og barskog ("Evergreen", terrengidentifikasjon 42).

Forskjellen mellom disse to skogstypene er at i en barskog er trærne eviggrønne i den forstand at de ikke mister nålene sine om vinteren. Trær i en løvskog har derimot blader som faller av om vinteren. I tillegg har NLCD datasettet definert en skogstype for blandet skog ("Mixed", terrengidentifikasjon 43). Denne skogstypen inneholder trær som både finnes i løv- og barskog.

For å ta hensyn til inndelingen av ulike skogstyper i vår modell må vi inkludere denne informasjonen i de ulike tretypene våre. Enhver tretype assosieres altså med en tilhørende skogstype som sørger for at løvtrær ikke blir utplassert i en rastercelle med terrengidentifikasjon barskog og omvendt. Dette skal vi komme tilbake til i seksjon 4.5.

4.1.2 Inndeling av rasteret i quadtreet

Vi må forlange at rasteret vi jobber med er kvadratisk: antall rader er lik antall kolonner. I tillegg må antall rader og kolonner være en potens av 2. Grunnen til dette er at vi ønsker å legge quadtreet oppå rasteret vårt. Rotnoden i quadtreet skal dekke over hele rasterdatasettet, og barna til rotnoden skal hver dekke 1/4 av rasteret. For å forsikre oss om at tiler på samme nivå spenner over like mange rasterceller i quadtreet, må vi kunne dele rasteret likt i begge retninger i planet. Om rasteret er kvadratisk og en potens av 2, innfrir vi dette kravet. Et eksempel på et quadtre og vårt NLCD rasterdatasett er gitt på figur 4.3.



Figur 4.3: *Her ser vi både vårt rasterdatasett og det tilhørende quadtreet.*

Vi innfører $(t_{i1}, t_{i2}, t_{j1}, t_{j2})$ som rastergrensene til en tile t. Vi bruker vanlig matrisenotasjon ved å la i være i'te rad og j være j'te kolonne. Siden en tile dypere ned i quadtreet spenner over færre celler i rasteret vårt, ønsker vi å vite indeksene til cellene som spenner over t. Om vi vet indeksene til cellene som ligger innenfor tilen t, slipper vi å lete igjennom hele rasteret når vi skal finne hvilken celle som inneholder et tilfeldig punkt *pos* i t. Denne problemstillingen er beskrevet i seksjon 4.1.3.

Vi finner rastergrensene $(t_{i1}, t_{i2}, t_{j1}, t_{j2})$ til en tile t på samme måte som vi beregnet $(t_{x_{min}}, t_{x_{maks}}, t_{y_{min}}, t_{y_{maks}})$. Denne fremgangsmetoden ble illustrert i algoritme 3.1 på side 15 og vi gjør tilsvarende prosess for å finne rastergrensene: del rasteret til foreldretilen i to for begge retninger i planet og sett $(t_{i1}, t_{i2}, t_{j1}, t_{j2})$ avhengig av orienteringen o_t til t. For å illustrere dette antar vi at vi skal finne rastergrensene til barna til rotnoden i quadtreet gitt vårt 256×256 NLCD rasterdatasett. Rotnodens rastergrenser er gitt ved:

$$(t_{i1}, t_{i2}, t_{j1}, t_{j2}) = (0, 256, 0, 256). \tag{4.1}$$

Vi halverer rasteret i begge retninger i planet for å finne i_Δ og j_Δ. Disse tilsvarer x_Δ og y_Δ i algoritme 3.1. Gitt rotnoden p finner vi i_Δ og j_Δ ved:

$$i_\Delta = \frac{1}{2}(p_{i2} - p_{i1}) = \frac{1}{2}(256 - 0) = 128, \tag{4.2}$$

$$j_\Delta = \frac{1}{2}(p_{j2} - p_{j1}) = \frac{1}{2}(256 - 0) = 128. \tag{4.3}$$

Dersom orienteringen o_t til barna t er inneholdt i den nordlige halvdel av rotnoden p setter vi:

$$(t_{i1}, t_{i2}) = (p_{i1}, p_{i1} + i_\Delta). \tag{4.4}$$

Ellers så er:

$$(t_{i1}, t_{i2}) = (p_{i1} + i_\Delta, p_{i2}). \tag{4.5}$$

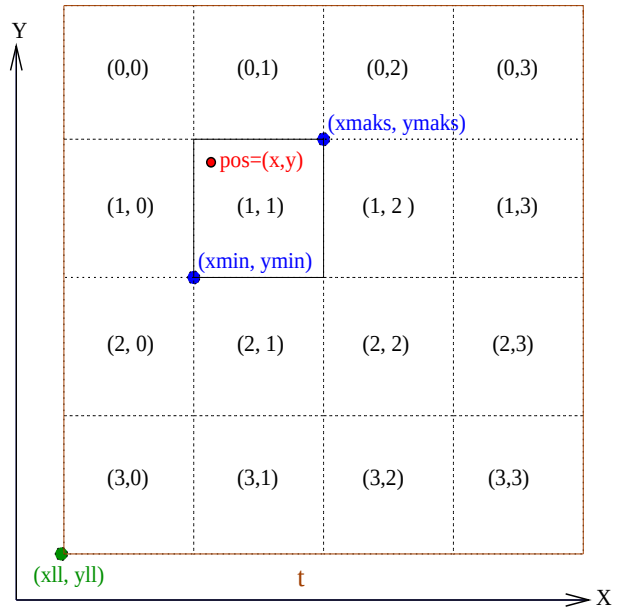
For å finne (t_{j1}, t_{j2}) undersøker vi om o_t er i den østlige halvdel av rotnoden som i algoritme 3.1. For barna til t vil rastergrensene til disse igjen være avhengig av rastergrensene til t. Siden beregningen av $(t_{i1}, t_{i2}, t_{j1}, t_{j2})$ er nesten identisk med algoritme 3.1, antar vi videre i denne oppgaven at vi har gitt rastergrensene $(t_{i1}, t_{i2}, t_{j1}, t_{j2})$ til enhver tile t i quadtreet.

4.1.3 Terrengidentifikasjonen til et tilfeldig punkt i en tile

NLCD datasettet gir også posisjonen til det nederste venstre hjørnet i rasteret. Vi kaller dette punktet for (x_{ll}, y_{ll}). For å finne grensene til en celle $c_{i,j}$ i rasteret, må vi finne avbildningsfunksjoner som for en gitt indeks (i, j) finner de tilhørende koordinatene i terrenget som $c_{i,j}$ spenner ut. Som før er i i'te rad og j j'te kolonne. I tillegg bruker vi vanlig praksis i programmeringsterminologi ved å la første celle i første rad i rasteret være $c_{0,0}$ og siste element i siste rad være $c_{N-1,N-1}$, der N er antall rader eller kolonner i rasteret. Vi innfører også c_s som oppløsningen på en celle i rasteret.

Grunnen til at vi ønsker å finne grensene til en celle $c_{i,j}$ er at vi må vite om et vilkårlig punkt $pos = (x, y)$ er inneholdt i $c_{i,j}$. Altså, gitt et punkt

pos innenfor en tile t i quadtreet, hvordan kan vi finne cellen $c_{i,j}$ i t som inneholder *pos*? For å besvare dette spørsmålet, må vi finne koordinatene (x_{min}, y_{min}) og (x_{maks}, y_{maks}) til en celle $c_{i,j}$. Denne problemstillingen er vist på figur 4.4 for $N = 4$.



Figur 4.4: *Et eksempel på et* 4×4 *raster som dekker over en tile* t*. Tallene i midten av hver celle angir hvilken* (i, j) *indekser som gir cellen* $c_{i,j}$*. Gitt et vilkårlig punkt* $pos = (x, y)$ *i* t*, må vi finne både* (x_{min}, y_{min}) *og* (x_{maks}, y_{maks}) *for alle* $c_{i,j}$ *innenfor* t*. Dette gjøres fordi vi ønsker å finne hvilken rastercelle* $c_{i,j}$ *som inneholder punktet pos.*

Det første vi merker oss er at det holder med å finne (x_{min}, y_{min}) for en celle $c_{i,j}$ som på figur 4.4 siden:

$$(x_{maks}, y_{maks}) = (x_{min} + c_s, y_{min} + c_s). \tag{4.6}$$

For å finne koordinatene til en celle $c_{i,j}$ i x-retningen, må vi se på verdien av j siden kolonnene ligger i x-retningen som på figur 4.4. For en gitt indeks j kan vi finne x_{min} til en vilkårlig celle $c_{i,j}$ ved:

$$x_{min} = x_{ll} + c_s j. \tag{4.7}$$

Å finne y_{min} for en celle $c_{i,j}$ er mer komplisert. Maksimal y_{min} verdi forekommer når $i = 0$. Etter hvert som indeksen i øker for en celle $c_{i,j}$, blir den tilsvarende y_{min} verdien for $c_{i,j}$ mindre som på figur 4.4. Ved å ta hensyn til dette kan vi finne y_{min} til $c_{i,j}$ for en gitt indeks i ved:

$$y_{min} = y_{ll} + c_s((N - 1) - i), \tag{4.8}$$

hvor N fortsatt er antall kolonner (eller rader siden rasteret er kvadratisk) i datasettet.

Vi er nå i stand til å finne ut koordinatene $(x_{min}, y_{min}, x_{maks}, y_{maks})$ til en celle $c_{i,j}$ gitt indeks (i, j). Å svare på om et punkt $pos = (x, y)$ ligger i $c_{i,j}$ består da kun om å sjekke at pos ligger innenfor grensene til $c_{i,j}$. Vi kan dermed si at $pos \in c_{i,j}$ dersom:

$$x_{min} \leq x \leq x_{maks}, \tag{4.9}$$

og

$$y_{min} \leq y \leq y_{maks}, \tag{4.10}$$

for et punkt $pos = (x, y)$.

La oss nå anta at vi har gitt et punkt pos og et rasterdatasett. Vi vet at rotnoden i quadtreet vil spenne over alle cellene, og at antall celler halveres i begge retninger for barna til rotnoden. For enkelhetens skyld sier vi at punktet pos skal plasseres i rotnoden i quadtreet. Vi får tilsvarende prosess for en vilkårlig tile t i quadtreet; eneste forskjell er mengden celler som spenner over t.

En mulig løsning på å finne hvilken celle $c_{i,j}$ som inneholder pos er å løpe igjennom alle cellene i rotnoden. For hver $c_{i,j}$ må vi altså undersøke om $pos \in c_{i,j}$, og stoppe letingen dersom riktig $c_{i,j}$ ble funnet. For store rasterdatasett, som for eksempel rasteret som vi arbeider med i denne oppgaven på 256×256 celler, vil verste tilfelle være iterasjonen over alle $256 \times 256 = 65536$ cellene i rasteret.

En bedre løsning er å bruke en variant av "Divide and Conquer" algoritmer. Disse kjennetegnes ved at man løser et stort problem ved å splitte problemet opp i flere små delproblemer som løses rekursivt [12, side 370]. I vårt tilfelle tilsvarer dette å finne riktige (i, j) indekser til en celle $c_{i,j}$ som inneholder et vilkårlig punkt pos. Dette problemet kan løses rekursivt ved å se på de to indeksene (i, j) hver for seg. Vi viser hvordan vi finner indeksen for j; tilsvarende fremgangsmåte gjøres for å finne riktig i.

For å finne riktig j indeks er det altså kolonnene vi må se på siden kolonnene ligger i x-retningen i planet. Tanken er at vi for hvert skritt i rekursjonen halverer antall kolonner vi må søke igjennom. Vi halverer mengden av kolonner i rasteret helt til vi står igjen med to kolonner; der punktet pos må eksistere i en av dem. Gitt x (x-verdien til punktet $pos = (x, y)$), A ($= t_{j1}$) og B ($= t_{j2}$), finner vi j-indeksen ved følgende algoritme:

Algoritme 4.1, finnCelleX(x, A, B)

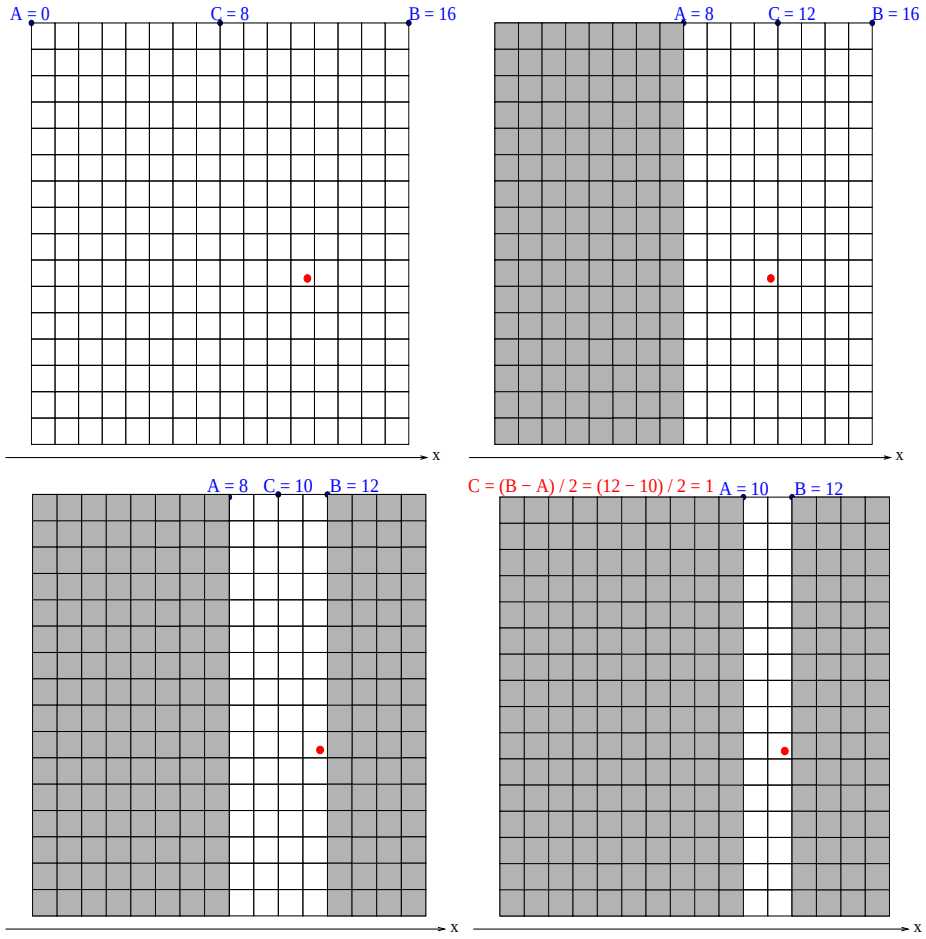
```
1  C = (B - A) / 2
2  x_check = x_ll + c_s(A + C)
3  hvis C != 1
4     hvis x ≤ x_check
5        finnCelleX( x, A, A + C)
6     ellers
7        finnCelleX( x, A + C, B )
8  ellers
9     x_1 = x_ll + c_s A
10    x_2 = x_1 + c_s
11    x_3 = x_2 + c_s
12    hvis x_1 ≤ x ≤ x_2
13       returner A
14    ellers
15       returner A + 1
```

Med et punkt $pos = (x, y)$, et 16×16 raster og rotnoden i quadtreet som spenner over alle 16×16 celler i rasteret, begynner vi med å kalle *finnCelleX(x, 0, 16)*. Prosessen med å finne j indeksen for dette tilfellet er vist på figur 4.5. Legg merke til at vi indekserer B med en indeks mer enn i rasterdatasettet vårt (husk at vi indekserer en celle $c_{i,j}$ i rasteret fra 0 til $N - 1$, hvor N i dette tilfellet er 16). Grunnen til at vi indekserer rasteret slik er for å gjøre beregningen av C på linje 1 i algoritme 4.1 så enkel som mulig.

Det første vi gjør er å finne C gitt som halvparten av antall kolonner $(B - A)$ som spenner over en tile t. Er $C \neq 1$, har vi mer enn to potensielle celler i kolonneretningen som kan inneholde punktet *pos*. Vi må da rekursivt halvere antall celler vi ser på helt til $C = 1$. På figur 4.5 øverste til venstre er $C = (16 - 0)/2 = 8$. Siden $C \neq 1$ må vi kalle *finnCelleX(...)* rekursivt. Skal vi kalle *finnCelleX(...)* rekursivt, må vi vite på hvilken side av C punktet *pos* ligger. For å undersøke dette må vi finne x-verdien til C ved ligning (4.7) med $j = A + C$. Vi kaller denne x-verdien for x_{check}.

Ligger p på venstre siden av x_{check}, kaller vi *finnCelleX(x, A, A + C)*. Ellers ligger x på høyresiden av x_{check} og vi kaller *finnCelleX*$(x, A + C, B)$. På første figuren øverst til venstre på figur 4.5 finner vi for eksempel ut at x ligger på høyresiden av x_{check}, og vi må derfor kalle *finnCelleX*$(x, 8, 16)$ $(A + C = 0 + 8 = 8)$.

Denne prosessen gjentas rekursivt helt til vi får $C = 1$ som nederst til høyre på figur 4.5. Siden vi vet at vi da står igjen med to kolonner, må punktet *pos* ligger i en av disse. Vi finner hvilken av de to kolonnene som inneholder *pos* ved å undersøke x-verdien til *pos* mot x-verdiene til kolonnene. Tilsvarende prosess gjøres for å finne i-indeksen til cellen $c_{i,j}$



Figur 4.5: *Eksempel på hvordan man effektivt finner riktig j indeks for en celle $c_{i,j}$ som inneholder et punkt pos (markert med en rød sirkel). Vi halverer rekursivt rasteret helt til $C = (B - A)/2 = 1$ som i tilfellet nederst til høyre. Dermed vet vi at pos må befinne seg i en av de to resterende cellene.*

som inneholder *pos* ved å kalle *finnCelleY(y, A, B)*, men her må $A = t_{i1}$ og $B = t_{i2}$.

Vi bruker dette når vi skal finne terrengidentifikasjonen til en celle $c_{i,j}$ som inneholder et vilkårlig punkt *pos*. Gitt $pos = (x, y)$ innenfor en tile t, samt rastergrensene til t, finner vi terrengidentifikasjonen id ved:

Algoritme 4.2, finnTerrengID($p, t_{i1}, t_{i2}, t_{j1}, t_{j2}$)

1 $j = finnCelleX(\ p.x,\ t_{j1},\ t_{j2}\ )$
2 $i = finnCelleY(\ p.y,\ t_{i1},\ t_{i2}\ )$
3 $id = c_{i,j}$.terrengidentifikasjon

4.1.4 Maksimalt nivå i quadtreet

En tile t på nivå l_t i quadtreet dekker over:

$$2^{(E-l_t)}, \tag{4.11}$$

celler i rasteret. Her er E gitt som eksponenten vi må opphøye 2 med for å få antall rader eller kolonner N i datasettet vårt. Vi finner dermed E ved:

$$E = log_2 N. \tag{4.12}$$

I vårt tilfelle, med et rasterdatasett på 256×256 celler, finner vi $E = 8$ ($log_2 256 = 8$). Det dypeste nivået en tile i quadtreet kan befinne seg på må dermed være på nivå $l_t = E$ siden ligning (4.11) gir:

$$2^{(E-l_t)} = 2^{(E-E)} = 2^0 = 1. \tag{4.13}$$

En tile t i quadtreet kan aldri spenne over mindre enn en celle i rasteret.

Vi ønsker at brukeren selv skal kunne gi det maksimale nivået til quadtreet. Vi innfører dermed l_{maks} som det brukerdefinerte maksimale nivået en tile i quadtreet kan befinne seg på. Ingen tiler på nivå l_{maks} skal ha barn. Det eneste vi må kreve er at:

$$l_{maks} \leq E \tag{4.14}$$

hvor E er som i ligning (4.12).

Grunnen til at vi ønsker å la brukeren selv gi en maksimal dybde l_{maks} er at ulike datasett kan ha ulik oppløsning på cellene. For eksempel kan det tenkes at dersom man har et datasett hvor oppløsningen på cellene er på et par meter, vil det kanskje ikke være ønskelig å instansiere tiler på det dypeste nivået. Det er da klart at ulike rasterdatasett fører til ulike visualiseringsbehov. For å lage skogmodellen vår så fleksibel som mulig er det opptil brukeren av applikasjonen å selv gi en passende verdi for l_{maks}. Dette gjøres i skogkonfigurasjonsfilen introdusert i tillegg A.

4.2 Oppbygging av skogmodellen

Siden vi ønsker sanntidsvisualisering må vi prøve å forhindre at for mange objekter, i vårt tilfelle trær, blir sendt ned til grafikkortet for prosessering som nevnt i seksjon 2.1. For å tilfredstille dette kravet ønsker vi å tegne flere og flere trær ettersom vi kommer nærmere og nærmere landskapet. Vi skal se på hvordan quadtreet kan brukes for å oppnå dette.

4.2.1 Treposisjoner nedover i quadtreet

La oss nå anta at vi har gitt en verdi for l_{maks}. Vi må nå se på hvordan vi skal utplasserer treposisjoner i quadtreet. Siden vi kan assosiere quadtreet med en synsavhengig LOD modell som nevnt i seksjon 3.2.1, vet vi at dypere nivåer i quadtreet tilsvarer finere oppløsning av terrenget. Jo nærmere vi befinner oss landskapet, jo dypere ned i quadtreet ønsker vi at vi skal traversere. Vi utplasserer treposisjoner i quadtreet slik at vi under visualiseringen ser stadig flere trær etterhvert som man traverserer dypere ned i quadtreet. På det dypeste nivået l_{maks} i quadtreet er dermed alle trærne i skogmodellen synlige.

Dette gjøres ved å først finne det maksimale antall treposisjoner som ønskes utplassert i rotnoden i quadtreet. Dette tallet er heretter kalt η. For hvert nivå l_t dypere ned i quadtreet ønsker vi å plassere ut færre og færre treposisjoner. Dette kommer av visualiseringshensyn: skal vi tegne treposisjonene i en tile t, skal vi i tillegg tegne alle treposisjonene som ble utplassert i foreldretilen. Foreldretilen skal igjen tegne alle treposisjonene som ble utplassert i sine foreldre og så videre oppover i quadtreet. Vi får dermed flere trær som må tegnes til skjerm jo dypere ned i quadtreet vi traverserer.

Vi søker en funksjon som, avhengig av det maksimale antall treposisjoner i rotnoden η og nivået l_t til en tile t, finner ut hvor mange treposisjoner som vi ønsker utplassert i t. Resultatet av denne funksjonen, kalt ω_{l_t}, er dermed det maksimale antallet treposisjoner som kan utplasseres i en tile t på nivå l_t:

$$\omega_{l_t} = \eta - \kappa l_t, \tag{4.15}$$

hvor κ er et tall som bestemmer hvor fort ω_{l_t} skal minskes for hvert tilenivå l_t nedover i quadtreet. For eksempel, ved å sette $\eta = 100$, $\kappa = 15$ og $l_t = 4$, får vi fra ligning (4.15):

$$\omega_4 = 100 - (15 * 4) = 40. \tag{4.16}$$

Med andre ord, det maksimale antall treposisjoner som kan plasseres i en tile på nivå 4 er 40.

Valget av en god η og κ i (4.15) er viktig fordi disse to variablene tilsammen styrer hvor mange treposisjoner som totalt utplasseres i landskapet. Velger vi en stor η og liten κ vil det, for hvert nivå l_t i quadtreet, bli utplassert et stort antall treposisjoner siden ω_l ikke minskes nok når vi traverserer dypere i treet. Dette vil i verste fall føre til at vi ikke oppnår sanntidsvisualisering, da mengden av treobjekter som må behandles av grafikkortet blir for stort. Om vi derimot velger en liten η og stor κ får vi få treposisjoner i de øverste tilene i quadtreet og ingen treposisjoner i de nederste. Vi får da en virtuell skog med svært spredte trær som fører til dårlig naturtro gjengivelse av en ekte skog.

La oss anta at vi har en verdi for det maksimale antall treposisjoner som kan utplasseres i rotnoden η. Hvordan skal vi velge en passende κ som sørger for å minske ω_l på en fornuftig måte? Svaret på dette spørsmålet henger sammen med det maksimale nivået til quadtreet l_{maks}. Gitt en l_{maks} vet vi at det ikke kan eksistere noen tiler på nivå $(l_{maks} + 1)$. Dermed bør $\omega_{(l_{maks}+1)} = 0$:

$$\omega_{(l_{maks}+1)} = \eta - \kappa(l_{maks} + 1) = 0. \tag{4.17}$$

Siden vi allerede vet (l_{maks+1}) og η, er dermed et godt valg for κ gitt ved

$$\kappa = \frac{\eta}{(l_{maks} + 1)}. \tag{4.18}$$

Med κ som i ligning (4.18), vil vi dermed sørge for at ω_l blir mindre og mindre for $l_t = 1, 2, ..., l_{maks}$. Vi unngår samtidig at ω_l blir negativ for dårlig valg av κ. Dermed er η og l_{maks} de eneste tallene som bestemmer hvor mange treposisjoner som skal utplasseres i quadtreet.

Akkurat som l_{maks}, lar vi også brukeren av modellen vår selv bestemme en passende verdi for det maksimale antall treposisjoner som kan utplasseres i rotnoden η. Grunnen til dette er blant annet:

- Ved å la brukeren selv bestemme hvor mange treposisjoner som skal plasseres ut i quadtreet, tar vi høyde for at ulike grafikkort klarer å prosessere ulike mengder med data i sanntid. En η verdi for skogmodellen som kjører på en gitt datamaskin vil ikke nødvendigvis gi like gode resultater på en annen datamaskin da ytelsen på grafikkakseleratoren bestemmer antall grafiske primitiver den klarer å prosessere i sanntid.

- Ulike rasterdatasett vil ha varierende mengder med skogdekning. For datasett med få celler av terrengidentifikasjon skog vil vi ikke trenge en så høy η som for et raster med stor dekning av skogceller.

Verdien av η settes sammen med l_{maks} i skogkonfigurasjonsfilen.

4.2.2 Generering av quadtreet

Vi antar at vi er gitt både η og l_{maks}. Dermed er vi klare for å generere selve quadtreet. For å vite hvor quadtreet befinner seg i rommet, må vi sende med koordinatene som rotnoden spenner over. Som nevnt i seksjon 4.1.3 spesifiserer NLCD datasettet kun koordinatene det til nederste venstre hjørnet (x_{ll}, y_{ll}) av rasteret. Dette tilsvarer $(t_{x_{min}}, t_{y_{min}})$ for rotnoden. Siden vi vet cellestørrelsen c_s for en enkel celle i vårt raster finner vi $(t_{x_{maks}}, t_{y_{maks}})$ for rotnoden ved:

$$(t_{x_{maks}}, t_{y_{maks}}) = (t_{x_{min}} + c_s N, t_{y_{min}} + c_s N), \tag{4.19}$$

hvor N enten er antall rader eller kolonner i rasteret. Gitt grensene til rotnoden, kan vi dermed instansiere denne ved følgende algoritme:

Algoritme 4.3, instansiering av rotnoden

```
1  l_t = 0
2  beregn κ ved ligning (4.18)
3  (t_{x_min}, t_{y_min}) = (x_ll, y_ll)
4  (t_{x_maks}, t_{y_maks}) = ( t_{x_min} + c_s N, t_{y_min} + c_s N )
5  forsøk å plassere ut η trær
```

Vi setter nivået l_t for rotnoden til 0 før vi beregner κ. Så setter vi grensene til rotnoden før vi forsøker å utplassere η treposisjoner i rotnoden. Vi skal se på utplasseringsalgoritmer for å finne treposisjonene innenfor en tile i seksjon 4.4.

Etter å ha instansiert rotnoden kan vi instansiere en vilkårlig tile t. Fra seksjon 3.1.1 vet vi at vi forbinder enhver tile t med en unik orientering o_t i forhold til foreldretilen p. Vi instansierer t ved følgende algoritme:

Algoritme 4.4, lagBarn(p, o_t)

```
1  l_t = p.l_t + 1
2  beregnGrenser( p, o_t )
3  beregn ω_{l_t} ved ligning (4.15).
4  Forsøk å utplassere ω_{l_t} trær
```

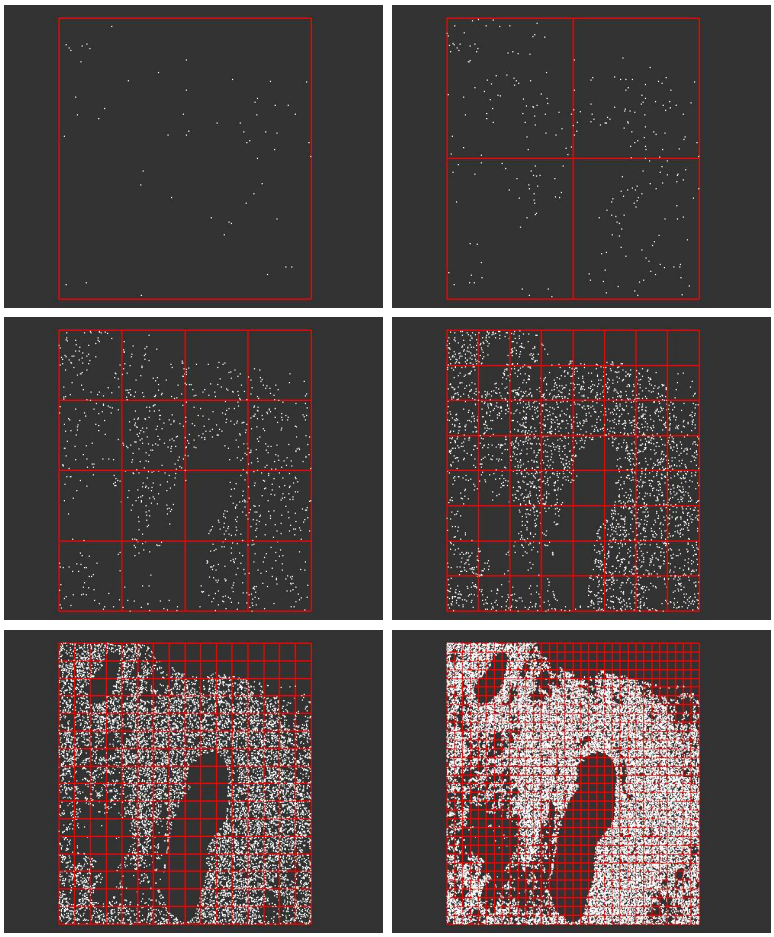
Vi starter med å sette nivået l_t for tilen t. Dette gjøres ved å sette l_t til en mer enn nivået til foreldretilen. Så beregner vi grensene som t spenner ut ved å kalle *beregnGrenser(...)* som i algoritme 3.1 på side 15. Vi beregner så ω_{l_t} før vi forsøker å utplassere ω_{l_t} treposisjoner i t. Den rekursive prosessen som genererer hele quadtreet kalles av rotnoden etter at vi har instansiert den ved følgende algoritme:

Algoritme 4.5, genererQuadtre(t) [I]

```
1  hvis l_t < l_maks
2     b = lagBarn( this, SV )
3     genererQuadTre( b )
4     b = lagBarn( this, S )
5     genererQuadTre( b )
6     b = lagBarn( this, N )
7     genererQuadTre( b )
8     b = lagBarn( this, NV )
9     genererQuadTre( b )
```

hvor de ulike orienteringen til t er de samme som på figur 3.2.

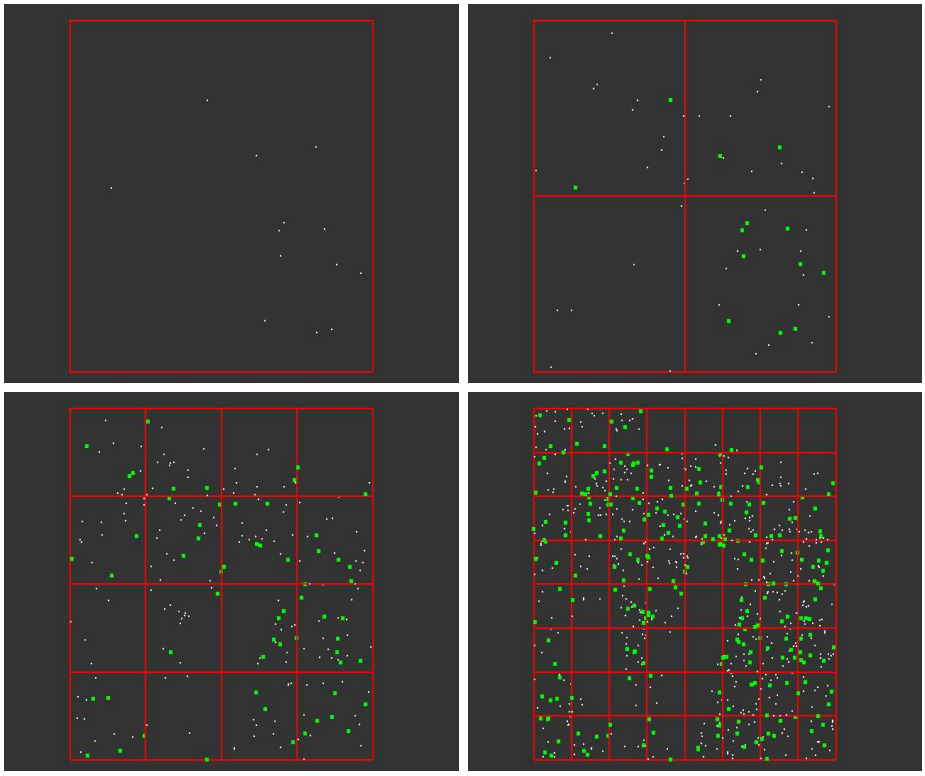
Prosessen vil kjøres rekursivt i de fire orienteringene til en tile helt til vi kommer ned til en tile som befinner seg på det dypeste quadtrenivået l_{maks}. Etter kjøringen av denne algoritmen er hele quadtreet generert. Et resultat av dette med vårt NLCD datasett er vist på figur 4.6.



Figur 4.6: *Et eksempel på quadtreet etter at algoritme 4.5 er ferdig. Her er* $l_{maks} = 5$ *og* $\eta = 100$. *Jo dypere ned i treet vi traverserer, jo flere treposisjoner blir synlige i landskapet. Dette er fordi vi tegner alle treposisjonene til foreldrene til* t *i tillegg til treposisjonen som ble utplassert i* t.

4.2.3 Duplisering av treposisjoner

En måte å forsikre oss om at all informasjonen til en foreldretile blir tatt vare i t er å duplisere treposisjonene i foreldretilen. Vi lar altså t inneholde både sine egne treposisjoner og treposisjonene til foreldretilen som ligger innenfor grensene til t. Denne prosessen er vist på figur 4.7.



Figur 4.7: *Et eksempel på duplisering av treposisjoner. Hvite prikker representerer treposisjonen som en tile t har utplassert; grønne prikker representerer treposisjoner som har blitt duplisert fra en foreldretile.*

Vi må altså duplisere alle treposisjonene til en foreldretile som ligger innenfor t etter at vi har forsøkt å plassere ut ω_{l_t} treposisjoner i t. Vi unngår dupliseringsprossesen for rotnoden i quadtreet da denne ikke har noen foreldre. Det er viktig å merke seg at duplisering av treposisjoner ikke er et krav i skogmodellen. Vi skal komme tilbake til dupliseringen av treposisjoner når vi snakker om visualisering av skogmodellen i kapittel 6. Grunnen til at vi nevner dette her er at om vi velger å la hver tile t inneholde lokale kopier av foreldretilens treposisjoner som ligger innenfor grensene til t, må dupliseringen nødvendigvis foregå under oppbygningen av quadtreet.

4.2.4 Avskjæringer

På figur 4.6 er det tydelig at det finnes flere tiler i quadtreet som er tomme i den forstand at de ikke inneholder noen treposisjoner. Dette skyldes at cellene i rasteret som er inneholdt i tilen ikke har terrengtype skog. Tomme tiler i quadtreet inneholder ikke noen relevant informasjon for skogmodellen. Videre vet vi at en quadtile sine barn spenner ut samme område som sine

foreldre. Dermed kan vi konkludere med at om tile t ikke inneholder noen treposisjoner, vil heller ikke barna til t inneholder noen treposisjoner.

Det er dermed unødvendig å instansiere en tile t dersom antall utplasserte treposisjoner i foreldretilen er 0. Vi innfører μ_t som antall utplasserte treposisjoner i en tile t. Dette tallet er gitt som resultatet av å forsøke å plassere ut ω_{l_t} treposisjoner i t. Vi må altså skille mellom det maksimale antall treposisjoner som kan utplasseres i en tile t (ω_{l_t}) og det faktiske antall treposisjoner som ble utplassert i t (μ_t). Vi skal komme tilbake til dette i seksjon 4.4.

For å ta hensyn til tomme tiler under quadgenereringen må vi forandre litt på algoritme 4.5 på side 34. Vi antar at vi har gitt antall utplasserte trær i en tile t. Den nye algoritmen for å generere quadtreet blir som følger:

Algoritme 4.6, genererQuadtre(t) [II]

```
1  hvis μ_t ≠ 0
2     hvis l_t < l_maks
3        t = lagBarn( this, SV )
4        genererQuadTre( t )
5        t = lagBarn( this, S )
6        genererQuadTre( t )
7        t = lagBarn( this, N )
8        genererQuadTre( t )
9        t = lagBarn( this, NV )
10       genererQuadTre( t )
```

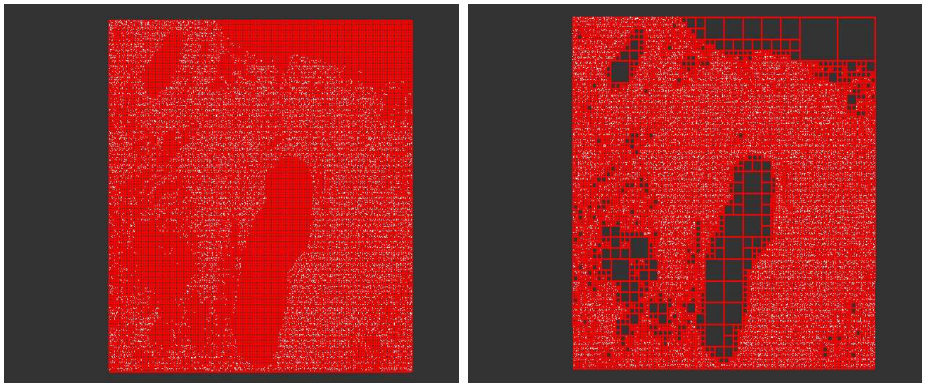
Den eneste forskjellen mellom algoritme 4.5 på side 34 og algoritme 4.6 er at vi kun lager barn til en tile t dersom antall utplasserte treposisjoner i t er forskjellig fra 0. Figur 4.8 viser effekten av å ikke instansiere barn til en tile som ikke inneholder noen trær. Differansen mellom antall quadtiler instansiert i dette tilfellet er $21845 - 17041 = 4804$. Dette tilsvarer en minsking av antall quadtiler på $\approx 21.99\%$ i dette tilfellet. Antall tiler som ikke må instansieres vil nødvendigvis variere avhengig av det gitte rasterdatasettet.

4.3 Traversering av quadtreet

Etter å ha generert quadtreet ønsker vi å traversere denne som en synsavhengig LOD modell som nevnt i seksjon 3.2.1. For å gjøre dette må vi introdusere feilmål. Dette sørger for at vi traverserer dypere ned i quadtreet jo nærmere landskapet vi er.

4.3.1 Quadtreet som en synsavhengig LOD modell

Vi må ha vite hvor dypt ned i quadtreet vi må traversere til enhver tid. Denne metoden må ta hensyn til to faktorer:



Figur 4.8: *Et eksempel på quadtreet for vårt* 256×256 *rasterdatasett med (til høyre) og uten (til venstre) avskjæringer. Her er* $l_{maks} = 7$ *og* $\eta = 100$.

- Størrelsen på en tile t. Dette kaller vi for *objektfeilen* δ_t.
- Avstanden fra det virtuelle kameraet til en tile t. Dette kaller vi for *avstandsmålet* ϵ_t.

Vi vil traversere dypere ned i quadtreet avhengig av avstanden fra en tile t til kameraet (ϵ_t). For å vite når vi er nærme nok t til at vi ønsker å traversere ned i barna til t må vi ha et mål som sier noe om t (δ_t). Innfører vi disse to målene og hele tiden undersøker hvor dypt vi skal traversere ned i datasettet under hver opptegning av en bilderamme oppnår vi en synsavhengig LOD modell. Figur 4.3 viser et eksempel på dette ved at vi har traversert dypere ned i quadtreet for tilene som er nærmest posisjonen til det virtuelle kameraet.

Vi begynner med å se på objektfeilen δ_t. Målet med δ_t er å assosiere et mål som kan knyttes opp mot hver tile t i quadtreet. Vi gir δ_t på formen:

$$\delta_t = \sigma_t K, \tag{4.20}$$

hvor σ_t er størrelsen på tilen t og K er en brukerdefinert konstant. Siden enhver tile t er kvadratisk kan σ_t beregnes ved å enten måle bredden eller høyden til t. Valget av K vil være med på å bestemme hvor dypt i treet vi må traversere gitt ϵ_t. Vi kommer tilbake til valg av K etter at vi har introdusert avstandsmålet ϵ_t.

Valget av ϵ_t må ta hensyn til både avstanden mellom en tile og kameraet og størrelsen til en piksel på skjermen. Et rent euklidsk avstandsmål kan introdusere kunstige effekter siden den ikke tar hensyn til at [11, side 88-90]:

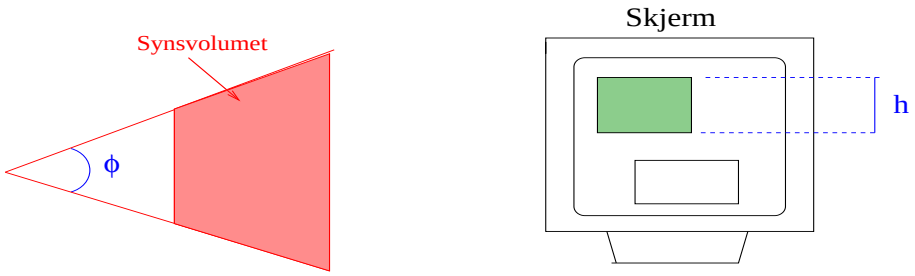
- Vinduet til applikasjonen kan endre form
- Synsvolumet kan endres

- Objektet kan bli skalert i en eller annen retning
- Oppløsningen på skjermen kan endres

Vi ønsker å vite hvor mye en piksel utgjør av det totale skjermbildet. Vi begynner med å finne vinkelen til en piksel θ:

$$\theta = \frac{\phi}{h}. \tag{4.21}$$

Her er h antall piksler i høyden på vinduet vårt på skjermen, og ϕ er synsvolumets "field of view"-vinkel. Denne vinkelen kan tenkes som en analogi til å velge kameralinse på et kamera: Å bruke en vid linse vil forvri objekter i en scene (særlig mot kantene av vinduet), mens en smal linse vil minske perspektivet og gå lenger inn i scenen. Både ϕ og h er vist på figur 4.9.



Figur 4.9: *ϕ er "field of view"-vinkelen knyttet til perspektiv projeksjonen. Ulike verdier for ϕ vil ha innvirkning på hvordan synsvolumet ser ut. Analogien til dette er bruken av ulik kameralinse for det virtuelle kameraet. Høyden h er antall piksler på vinduet til applikasjonen på skjermen.*

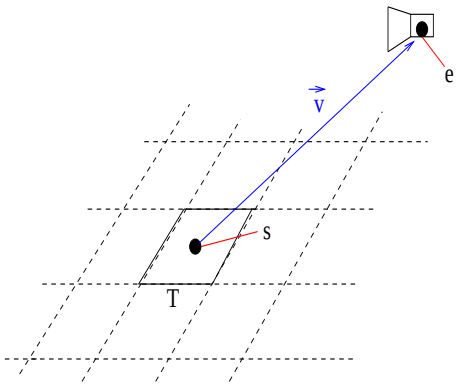
Fra ligning (4.21) ser vi at θ kun endrer seg hvis man endrer størrelsen på vinduet på skjermen eller synsvolumet. Dermed holder det å beregne θ en gang under oppstarten av applikasjonen vår og deretter kun når vinduet eller synsvolumet endres. Etter å ha beregnet θ må vi finne avstanden fra det virtuelle kamera til en tile.

For å beregne avstanden til en tile t må man først finne midtpunktet s i t. Dette punktet kan beregnes under instansiering av t. Vi antar at det virtuelle kameraet befinner seg på posisjonen e i rommet. Avstanden fra det virtuelle kameraet til en tile t er gitt som avstanden fra s til e som på figur 4.10.

Vi finner vektoren $\vec{\mathbf{v}}$ fra s til e ved:

$$\vec{\mathbf{v}} = (e \vec{-} s) = (e_x - s_x, e_y - s_y, e_z - s_z) = (v_x, v_y, v_z). \tag{4.22}$$

Den euklidske avstanden d til $\vec{\mathbf{v}}$ er gitt som:



Figur 4.10: *Vektoren* $\vec{\mathbf{v}}$ *er avstanden fra* s *til* e*. Lengden til* $\vec{\mathbf{v}}$ *gir avstanden fra en tile* T *til det virtuelle kameraet.*

$$d = \sqrt{\vec{\mathbf{v}}^2} = \sqrt{v_x^2 + v_y^2 + v_z^2}. \tag{4.23}$$

Etter å ha beregnet d er vi klare for å gi vårt avstandsmål ϵ_t som:

$$\epsilon_t = \theta d, \tag{4.24}$$

hvor θ er gitt som i ligning (4.21) og d er som i ligning (4.23).

Etter å ha funnet våre to feilmål δ_t og ϵ_t er vi klare for å traversere quadtreet som en synsavhengig LOD modell. Dette gjøres ved følgende algoritme:

Algoritme 4.7, traversere(t) [l]

```
1 beregn ε_t ved ligning (4.24)
2 hvis μ_t ≠ 0
3    hvis δ_t > ε_t og l_t ≤ l_maks
4       for alle barn b til t
5          traversere( b )
```

Vi kaller algoritme 4.7 med rotnoden for hver opptegning av en bilderamme. Algoritmen vil rekursivt traversere quadtreet som en synsavhengig LOD modell.

For hver tile t beregner vi avstandsmålet ϵ_t fra t til kameraet. Siden avstanden fra t til kameraet kan variere fra bilderamme til bilderamme, må ϵ_t alltid beregnes for hver t under traverseringen. Dette er i motsetning til objektfeilen δ_t som er konstant for hver tile t og kan dermed beregnes under instansieringen av t.

Hvis antallet utplasserte trær $\mu_t \neq 0$ for en tile t må vi undersøke om:

- $\delta_t > \epsilon_t$: Siden ϵ_t måler avstanden fra en tile t til det virtuelle kameraet, vil ϵ_t bli mindre etterhvert som kameraet nærmer seg t. Dermed vil objektfeilen δ_t for en tile t bli større enn ϵ_t så lenge man er nærme nok t. Så lenge $\epsilon_t \geq \delta_t$ skal vi ikke traversere dypere ned i t. Med en gang $\epsilon_t < \delta_t$, det vil si med en gang kameraet er nærme nok t, ønsker vi å traversere videre ned i barna til t.
- $l_t \leq l_{maks}$: Vi må forhindre at vi prøver å traversere for dypt i treet fordi tiler på det maksimale nivået $l_t = l_{maks}$ i quadtreet ikke har noen barn. Dermed vil algoritme 4.8 feile uten denne spørringen.

Vi stanser traverseringen med en gang vi enten har nådd maksimaldybden til treet l_{maks} eller at objektfeilen δ_t for en gitt tile er større eller lik avstandsmålet ϵ_t.

Vi husker fra ligning (4.20) at objektfeilen δ_t er avhengig av en brukerdefinert konstant K. Sett at vi nå har en konstant verdi for ϵ_t, eller sagt på en annen måte: det virtuelle kameraet står stille i vår scene. Siden høye verdier for K også vil gi høye verdier for δ_t må man traversere dypere ned i quadtreet før man endelig finner en tile t der $\epsilon_t \geq \delta_t$. Lav verdi for K vil føre til at traverseringen stopper tidligere fordi objektfeilen δ_t vil være liten.

Figur 4.11 illustrerer traverseringen av quadtreet for ulike verdier av K. Siden valget av K styrer hvor dypt vi skal traversere quadtreet for en gitt ϵ_t lar vi også K være gitt av brukeren i skogkonfigurasjonsfilen. Dermed kan brukeren selv bestemme ønsket detaljgrad i landskapet for en gitt avstand.

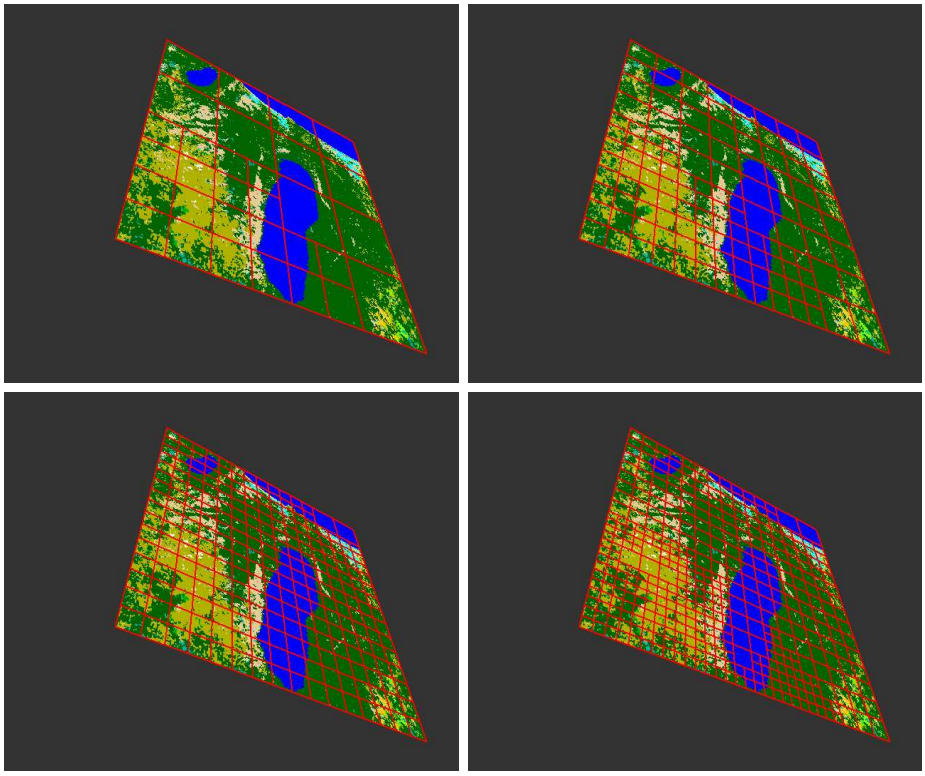
4.3.2 Synlighetsspørring under traversering

Som nevnt i seksjon 3.1.2 er det svært effektivt å utnytte egenskapene til quadtreet for å foreta hierarkisk synlighetsspørringer. Dette gjøres ved å assosiere en omringet sfære til enhver tile t i quadtreet. Ved å undersøke om sfæren er synlig innenfor synsvolumet kan vi finne ut om vi behøver å tegne en tile.

La nå B_t være sfæren som omringer en tile t. Vi får da følgende traverseringsalgoritme:

Algoritme 4.8, traversere(t) [II]

```
1  beregn ε_t
2  hvis B_t ikke befinner seg innenfor synsvolumet
3      avslutt traverseringen for t
4  hvis μ_t ≠ 0
5      hvis δ_t > ε_t og l_t ≤ l_maks
6          for alle barn b til t
7              traversere( b )
```



Figur 4.11: *Traversering av quadtreet med algoritme 4.8 for ulike verdier av* K*. Øverst til venstre er* $K = 0.51$*, øverst til høyre er* $K = 1.01$*, nederst til venstre er* $K = 1.51$ *og til slutt nederst til høyre* $K = 2.01$*. Kameraet er plassert på samme sted for alle figurene, slik at* ϵ_t *er den samme for alle tiler* t *i quadtreet.*

Den eneste endringen fra algoritme 4.7 på side 40 er innføringen av synlighetsspørringen i linje 2 og 3. Hvis sfæren B_t til en tile t ikke er synlig, er det unødvendig å traversere videre ned i barna til t som nevnt i seksjon 3.1.2. Vi returnerer derfor fra den rekursive traverseringsmetoden vår. Dette sparer grafikkakseleratoren for unødvendig prosessering av geometrisk informasjon som allikevel ikke bidrar til det ferdige bildet som vi ser på skjermen.

4.4 Utplasseringsalgoritmer

Vi har til nå ignorert hvordan vi utplasserer trær i en tile t i quadtreet. I denne oppgaven har vi sett på to aktuelle utplasseringsalgoritmer: utplassering med spredningsfaktor og utplassering uten spredningsfaktor. Begge algoritmene ble implementert for så å sammenligne resultatet av ulike kjøringer.

4.4.1 Utplassering av trær med spredningsfaktor

Spredningsfaktoren, herved betegnet c_t, er et tall som angir hvor mange av de underliggende rastercellene i hver tile t som har terrengtype skog. Beregningen av c_t er gitt som følger: for hver tile t i quadtreet summerer vi opp antall rasterceller i t som har terrengtype skog. Summen vi får deles så på antall rasterceller utspent av denne quadtilen. Vi får da en verdi mellom 0 og 1 hvor 1 angir at hele tilen er dekket av skogceller. Får vi 0 vet vi at ingen trær skal utplasseres i denne tilen fordi ingen rasterceller innenfor tilen er av terrengtype skog. Dette kan for eksempel forekomme hvis hele tilen dekker et område med vann. Verdier mellom 0 og 1 svarer dermed til at noen celler i det underliggende landskapsrasteret har terrengtype skog mens andre celler ikke har det. Algoritmen for å beregne spredningsfaktoren c_t for en tile t er gitt som:

Algoritme 4.9, beregnSpredningsfaktor(t)

```
1  c_t = 0
2  for alle rasterceller c_{i,j} som spenner ut t
3      hvis c_{i,j}.terrengidentifikasjon er av type skog
4          c_t = c_t + 1
5  c_t = c_t / R
```

hvor R er antall celler i rasteret som er inneholdt i tilen t.

Etter at vi har beregnet c_t bruker vi denne til å gi oss et estimat på hvor mange trær vi ønsker utplassert i hver tile. Vi beholder ω_{l_t} som i ligning (4.15) men nå er det maksimale antallet treposisjoner som kan utplasseres i en tile også avhengig av spredningsfaktoren c_t. Dette tallet, kalt Ω_t, beregnes ved å multiplisere c_t og ω_{l_t} for hver tile i quadtreet:

$$\Omega_t = c_t \omega_{l_t}. \tag{4.25}$$

Vi får dermed et nytt mål på det maksimale antall treposisjoner som kan utplasseres i en tile t. Vi gir utplasseringsalgoritmen som:

*Algoritme 4.10, utplassering **med** spredningsfaktor*

```
1  beregnSpredningsfaktor( t )
2  Beregn Ω_t ved ligning (4.25)
3  μ_t = 0
4  så lenge μ_t < Ω_t
5      finn et tilfeldig punkt pos i t
6      finnTerrengID( pos, t_i1, t_i2, t_j1, t_j2 )
7      hvis id == terrengtype skog
8          leggTilTre( pos, id )
9          μ_t++
```

Vi begynner med å beregne spredningsfaktoren c_t for tilen t som i algoritme 4.9 før vi beregner Ω_t. Siden vi har enda ikke utplassert noen treposisjoner i t setter vi antallet utplasserte treposisjoner i t, μ_t, til 0 på linje 3. Så starter løkken med å utplassere Ω_t treposisjoner i t.

Vi finner et tilfeldig punkt *pos* innenfor t. Så kaller vi *finnTerrengID(...)* som ved algoritme 4.2 på side 30. Her får vi altså bruk for metoden introdusert i seksjon 4.1.1 for å finne terrengidentifikasjonen *id* til rastercellen som inneholder punktet *pos*. Vi undersøker videre om *id* er en skogcelle. Er punktet *pos* inneholdt i en rastercelle med terrengidentifikasjon skog, legger vi *pos* inn i tilen t ved *leggTilTre(...)* metoden introdusert i seksjon 4.5. Vi har da funnet en treposisjon i tilen t og må øke antall utplasserte trær μ_t i t.

Om *pos* ikke ligger i en skogcelle prøver vi å finne en ny posisjon. Dette gjøres helt til vi finner en treposisjon som ligger innenfor en celle med terrengidentifikasjon skog. Algoritme 4.10 garanterer dermed at man alltid utplasserer Ω_t treposisjoner i tilen t siden utplasseringsløkken ikke slutter før $\mu_t = \Omega_t$.

4.4.2 Utplassering av trær uten spredningsfaktor

Den andre utplasseringsalgoritmen er ikke avhengig av å beregne en tilhørende spredningsfaktor for en tile t. I stedet ønsker vi å forsøke å utplassere ω_{l_t} treposisjoner som ved ligning (4.15) for t på nivå l_t. Algoritmen blir da som følger:

*Algoritme 4.11, utplassering **uten** spredningsfaktor*

```
1  n = 0
2  μ_t = 0
3  så lenge n < ω_{l_t}
4      finn et tilfeldig punkt pos i t
5      finnTerrengID( pos, t_i1, t_i2, t_j1, t_j2 )
6      hvis id == terrengtype skog
7          leggTilTre( pos, id )
8          μ_t++
9      n++
```

Vi ser av algoritme 4.11 at hvis punktet *pos* ikke ligger i en rastercelle med terrengidentifikasjon skog, gjør vi ikke annet enn å "hoppe" over dette forsøket. Vi bruker n i algoritme 4.11 som antall forsøkte utplasserte trær i tilen t. Etter å ha satt n til 0 på linje 1, øker vi n på linje 9 for hver iterasjon av utplasseringsløkken uavhengig om *pos* ligger innenfor en skogcelle eller ikke. Har vi funnet en treposisjon som ligger innenfor en skogcelle skal vi, som før, legge til *pos* i t. Siden vi har klart å utplassere en treposisjon i t må vi også øke antall utplasserte trær μ_t i t.

4.4.3 Sammenligning av utplasseringsalgoritmer

En ulempe med utplasseringsalgoritmen med spredningsfaktor er selve beregningen av denne. Siden vi må løpe igjennom alle de underliggende cellene i rasteret, kan dette være en tidkrevende prosess for store rasterdatasett. Fordelen er at om vi allikevel ikke skal utplassere trær i t ($\Omega_t = 0$) vil algoritmen tidlig forhindre unødvendig tidsforbruk i utplasseringsløkken.

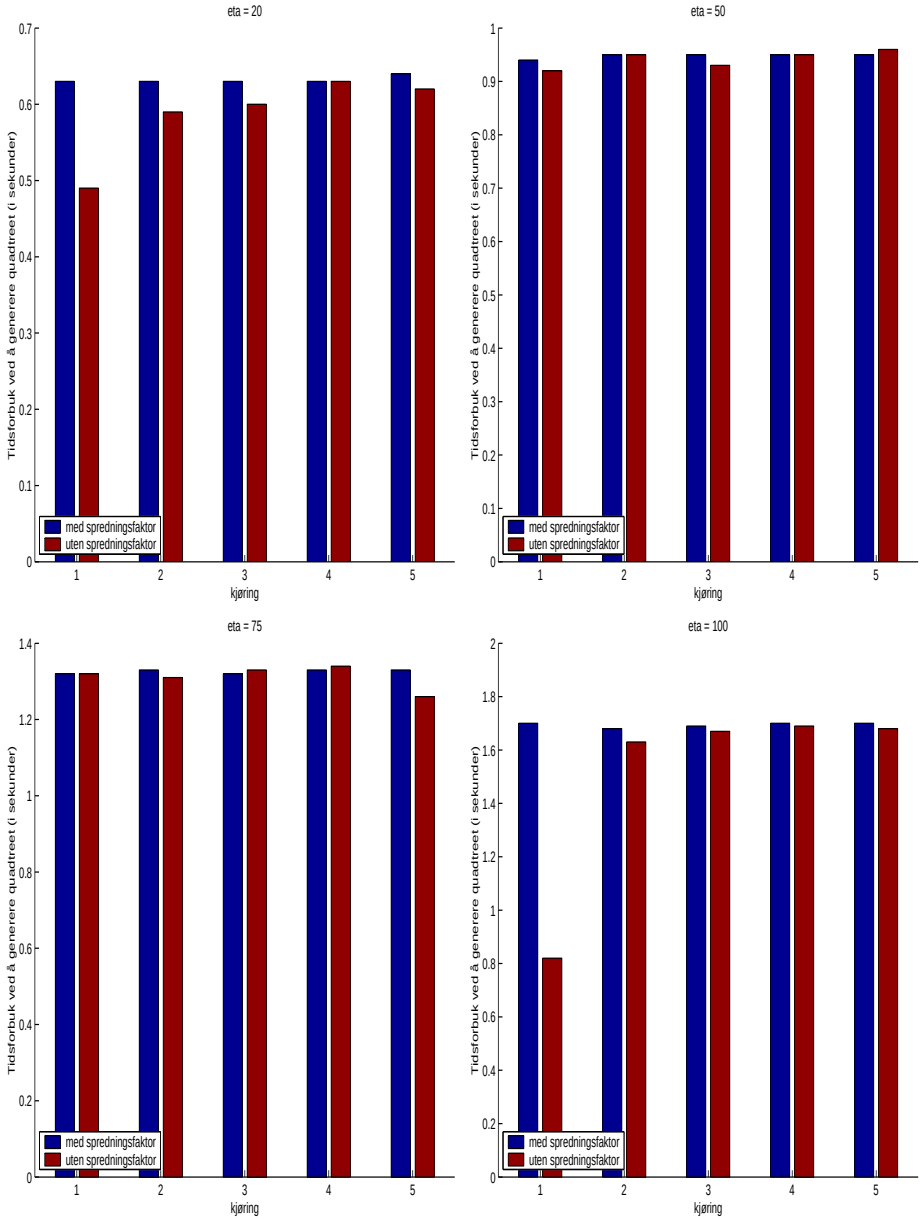
Sett nå at vi skal utplassere få trær i en tile t for utplasseringsalgoritmen med spredningsfaktor. Dette tilsvarer at vi har beregnet en lav verdi for Ω_t. Sagt på en annen måte: det finnes få celler i det underliggende rasterer til t som har terrengidentifikasjon skog. Vi har allerede sett at algoritme 4.10 på side 43 tvinger en tile t til å utplassere Ω_t trær. Siden en posisjonen i terrenget velges vilkårlig kan vi regne med at tidsforbruket for å finne en *pos* som ligger innenfor en celle med riktig terrengtype i verste fall kan være veldig stor.

Ulempen med utplasseringsalgoritmen uten spredningsfaktor er at vi ikke har noen forhåndsinformasjon om det underliggende rasteret. Siden vi ikke vet noe om cellene i rasteret som spenner ut en tile, kan vi ikke utnytte denne informasjonen for å foreta avskjæringer. Vi må dermed alltid forsøke å utplassere ω_{l_t} trær i enhver tile t. Fordelen med dette er at vi aldri tvinger t til å utplassere et gitt antall trær. En annen fordel er at ω_{l_t} vil nesten alltid være større enn Ω_t, siden vi ikke multipliserer med en spredningsfaktoren. Vi får dermed forsøkt utplassert flere treposisjoner med utplasseringsalgoritmen uten spredningsfaktor.

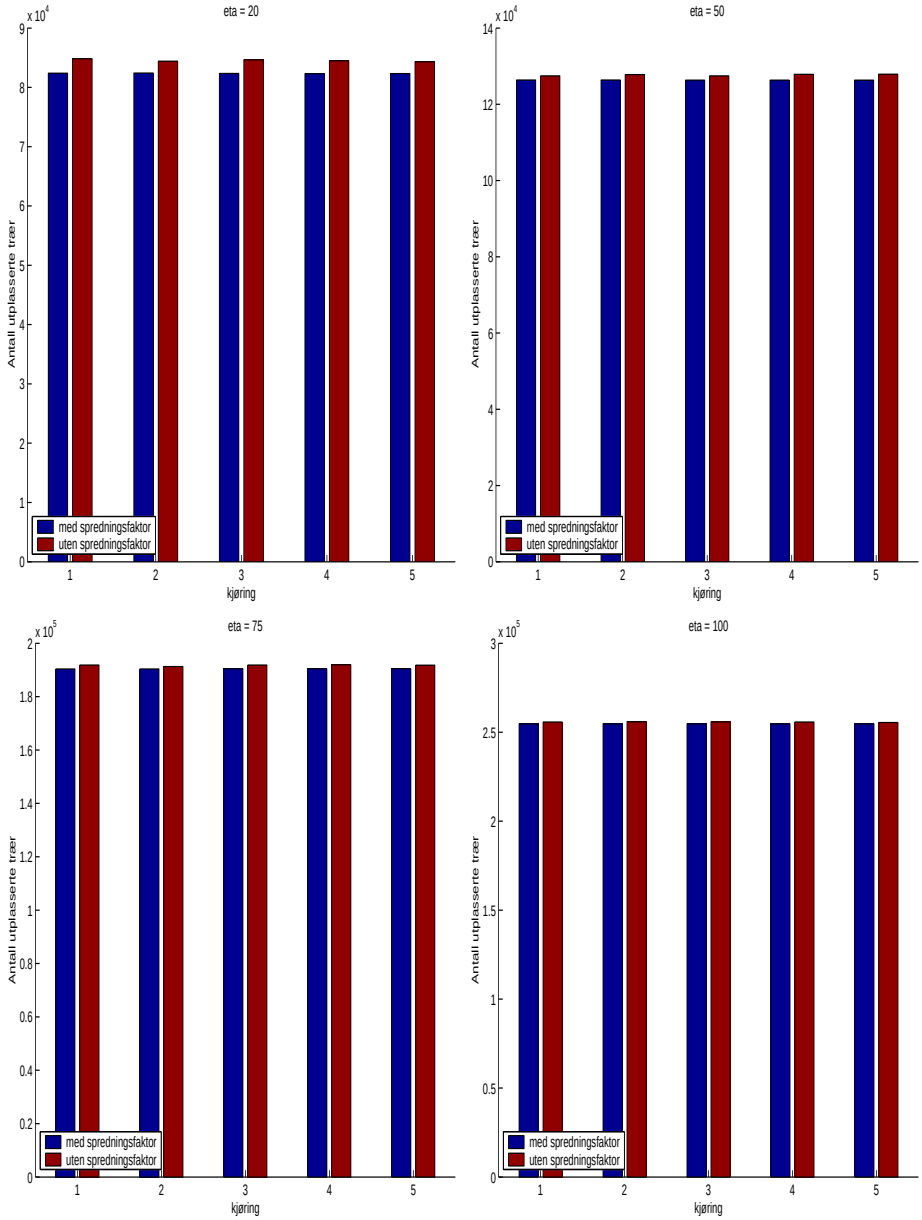
Figur 4.12 viser resultatet av tidsforbruket ved 5 ulike kjøringer med begge utplasseringsalgoritmer. For å få best mulig sammenligningsresultat måler vi resultatene etter økende antall utplasserte trær. Vi husker at η, det maksimale antall trær vi kunne utplassere i rotnoden, styrer dette. Målingene er gjort for 4 ulike verdier for η: $\eta = 20$, $\eta = 50$, $\eta = 75$ og $\eta = 100$. For alle tilfeller er det maksimale nivået i quadtreet $l_{maks} = 7$.

Som ventet er utplasseringsalgoritmen uten spredningsfaktor stort sett raskere. Dette skyldes både at algoritmen ikke må preberegne spredningsfaktoren c_t, samt at den ikke tvinger en tile til å utplassere et gitt antall trær. Størst avvik er det for første kjøring med $\eta = 100$ på figur 4.12. Her er utplasseringsalgoritmen uten spredningsfaktor nesten ett sekund raskere. Også første kjøring med $\eta = 20$ gir merkbar forskjell mellom våre to utplasseringsalgoritmer. Merk at noen kjøringer gir at utplasseringsalgoritmen med spredningsfaktor er raskere. Grunnen til at målingene varierer er at vi bruker vilkårlig valgte tall for å finne treposisjoner. Forskjellen i tidsforbruk er for det meste kun snakk om tiendels sekunder.

Vi måler også antall treposisjoner utplassert i landskapet. Figur 4.13 viser resultatet av antall utplasserte trær for vårt rasterdatasett. Som før har vi målt antall utplasserte treposisjoner til 5 kjøringer for fire ulike η verdier. Som ventet viser figur 4.13 at flere treposisjoner blir utplassert i landskapet

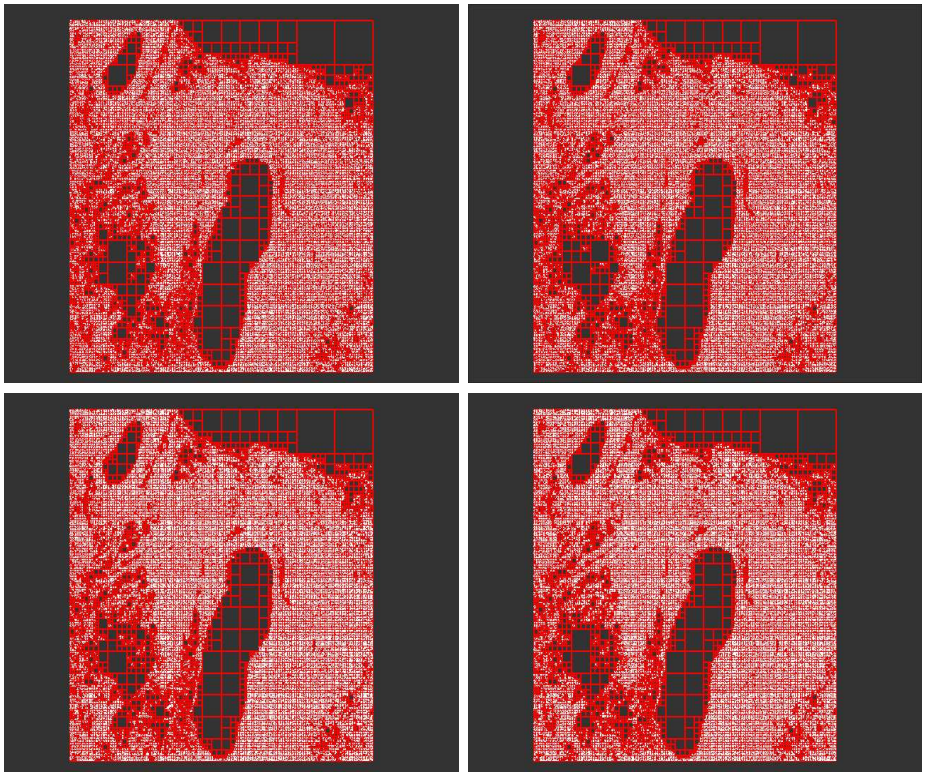


Figur 4.12: *Resultatet av tidsforbruket til* 5 *separate kjøringer for fire* η *verdier. Blå angir utplasseringsalgoritmen med spredningsfaktor og rød angir utplasseringsalgoritmen uten spredningsfaktor. Øverst til venstre er tilfellet hvor* $\eta = 20$*, øverst til høyre tilfellet hvor* $\eta = 50$*. Nederst til venstre er testkjøringene med* $\eta = 75$ *og nederst til høyre er* $\eta = 100$*. For alle kjøringer er* $l_{maks} = 7$*.*



Figur 4.13: *Resultatet av antall utplasserte trær til* 5 *kjøringer for fire* η *verdier. Blå angir utplasseringsalgoritmen med spredningsfaktor og rød angir utplasseringsalgoritmen uten spredningsfaktor. Øverst til venstre er tilfellet hvor* $\eta = 20$*, øverst til høyre tilfellet hvor* $\eta = 50$*. Nederst til venstre er testkjøringene med* $\eta = 75$ *og nederst til høyre er* $\eta = 100$*. For alle kjøringer er* $l_{maks} = 7$*.*

for alle kjøringer med utplasseringsalgoritmen uten spredningsfaktor. I snitt blir cirka 1000 flere trær utplassert med denne algoritmen. Til slutt ser vi på det visuelle resultatet av begge algoritmene vist på figur 4.14.



Figur 4.14: *Sammenligning av det visuelle resultatet av våre to utplasseringsalgoritmer. Venstre kolonne er resultatet etter utplasseringsalgoritmen med spredningsfaktor; høyre kolonne er resultatet av utplasseringsalgoritmen uten spredningsfaktor. Første rad er tilfellet* $\eta = 75$*, andre rad er tilfellet* $\eta = 100$*.*

Som vi ser er det minimal visuell forskjell mellom de to utplasseringsalgoritmene. Konsentrerer vi oss om området ved siden av vannet nederst til venstre på figurene ser vi for tilfellet $\eta = 100$ at utplasseringsalgoritmen med spredningsfaktor har klart å utplassere flere treposisjoner innenfor dette området enn for utplasseringsalgoritmen uten spredningsfaktor. Forskjellen er likevel så minimal at vi velger å se bort ifra denne.

Ut i fra dette har vi valgt utplasseringsalgoritmen uten spredningsfaktor som utplasseringsalgoritme for skogmodellen siden:

- Den bruker generelt mindre tid.
- Den utplasserer flere trær for en gitt η.

- Den gir minimal visuell forskjell fra utplasseringsalgoritmen med spredningsfaktor.

4.5 Treposisjoner og ulike tretyper

Etter å ha bestemt oss for utplasseringsalgoritme skal vi nå konsentrere oss om hvordan vi utplasserer ulike tretyper i landskapet. Som nevnt i seksjon 4.1.1 har vi for NLCD datasettet tre ulike skogstyper: løvskog, barskog og blandet skog. For å ta hensyn til dette ønsker vi nå at hver tretype i skogmodellen skal forbindes med en tilhørende skogstype.

4.5.1 Fordeling av ulike trær

Vi begynner med å se på *type* variabelen definert for hver tretype i skogkonfigurasjonsfilen introdusert i tillegg A. Variabelen angir om tretypen tilhører bar- eller løvskog. En tretype som har satt *type* variabelen til løvskog kan da ikke utplasseres i en rastercelle med terrengidentifikasjon barskog og omvendt. Alle tretyper, uavhengig om de tilhører bar- eller løvskog, kan utplasseres i en rastercelle med terrengidentifikasjon blandet skog.

Vi ønsker å angi et mål på hvor stor spredning det er av denne tretypen i terrenget. Spredningen til en tretype bestemmes av *spread* variabelen for hver tretype i skogkonfigurasjonsfilen og er gitt som ønsket prosentvis dekning av tretypen i en rastercelle med terrengidentifikasjonen lik tretypens tilhørende skogstype. Siden vi ønsker å angi prosentvis dekning må summen av *spread* variablene til alle tretypene for hver definerte skogstype i datasettet være 100.

For å illustrere de ulike variablene som beskriver en tretype ser vi på et spesifikt eksempel hentet fra skogkonfigurasjonsfilen i tillegg A:

```
name : M_TREE4
spread : 60
type : evergreen
```

For M_TREE4 tretypen er spread variabelen satt til 60. Vi ønsker altså at denne tretypen skal dekke $\approx 60\%$ av alle utplasserte trær i en rastercelle av terrengidentifikasjon barskog.

Enhver tretype T i skogmodellen vår er forbundet med to variable s_{start} og s_{slutt} som gir *spredning-grensene* til T gitt ved følgende algoritme:

Algoritme 4.12, beregnSpredningGrenser()

```
1 for alle skogstyper definert i rasterdatasett
2     s_1 = s_2 = 0
3     for alle tretyper T hvor T.type tilsvarer skogstypen
4         s_2 = s_2 + T.spread
5         T.s_start = s_1
6         T.s_slutt = s_2
7         s_1 = s_2 + 1
```

Spredning-grensene for alle tretypene blir satt med en gang applikasjonen har lest inn skogkonfigurasjonsfilen. Vi starter med å løpe igjennom de ulike skogstypene definert for rasterdatasettet. For alle tretypene gitt i skogkonfigurasjonsfilen med *type* lik denne skogstypen beregner vi en tilhørende s_{start} og s_{end} verdi. Disse brukes til å bestemme hvilken av våre gitte tretyper som skal velges for hver utplasserte treposisjon i terrenget.

Resultatet av algoritme 4.12 illustreres best med et eksempel. Tabellen under viser alle tretypene gitt i skogkonfigurasjonsfilen i tillegg A på side 123. Første kolonne er navnet på tretypen, andre kolonne gir den tilhørende skogstypen, tredje kolonne er verdien av spread som gitt i skogkonfigurasjonsfilen, og fjerde og femte kolonne viser verdien av de tilhørende spreadgrensene for den respektive tretypen etter at skogkonfigurasjonsfilen er lest inn og algoritme 4.12 er ferdig:

Tretype T	$T.type$	$T.spread$	$T.s_{start}$	$T.s_{end}$
M_TREE4	evergreen	60	0	60
M_TREE5	evergreen	30	61	90
M_TREE6	evergreen	10	91	100
P_WILLOW	deciduous	70	0	70
M_TREE1	deciduous	30	71	100

Gitt spredning-grensene for alle tretypene våre, er vi nå klare for å knytte enhver treposisjon i landskapet med en tilhørende tretype. La *pos* være en treposisjon i en tile t som er innenfor en rastercelle med terrengidentifikasjon *id*. Videre antar vi at *id* er en av våre tre skogstyper for NLCD datasettet slik at linje 6 i algoritme 4.11 på side 44 er sann. Vi skal dermed legge til *pos* i t ved *leggTilTre(...)* gitt ved følgende algoritme:

Algoritme 4.13, leggTilTre(pos, id)

```
1  hvis id == barskog eller id == løvskog
2      finn et vilkårlig tall r mellom 0 og 100
3      for alle tretyper T med T.type == id
4          hvis r ≥ T.s_start og r ≤ T.s_end
5              instansier et TrePosisjon objekt P
6              P.posisjon = pos
7              P.tretype = T
8              legg til P i tilen
9  ellers hvis id == blandet skog
10     finn et vilkårlig tall r mellom 0 og (T_N − 1)
11     instansier et TrePosisjon objekt P
12     P.posisjon = pos
13     P.tretype = T
14     legg til P i tilen
```

Hver treposisjon i en tile t er et objekt av klassen *TrePosisjon*. Vi kommer tilbake til alle variablene til denne klassen i seksjon 5.3, men kan nevne at et *TrePosisjon* objekt inneholder, foruten selve de geometriske koordinatene til treposisjonen, sin tilhørende tretype. Målet med algoritme 4.14 er dermed å knytte et *TrePosisjon* objekt til en tile t.

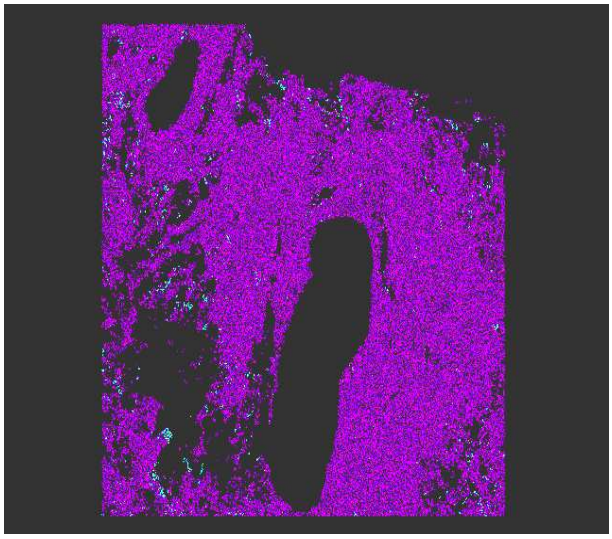
Vi begynner med å undersøke verdien til id for å finne hvilken skogstype rastercellen som inneholder *pos* har. For id lik bar- eller løvskog er prosessen den samme: Finn et vilkårlig tall r mellom 0 og 100 og undersøk hvilken tretype T av med terrengidentifikasjon id som har sine spredning-grenser innenfor r. Etter å ha funnet tretypen T, instansierer vi et nytt objekt av *TrePosisjon* klassen og setter det nye objektets posisjon til *pos* og tretypen til T.

I tilfellet hvor id er av terrengidentifikasjon blandet skog må vi forandre fremgangsmåten. Dette må gjøres siden alle tretyper kan utplasseres i rastercellen uavhengig av skogstypen. Vi innfører T_N som antall tretyper gitt i skogkonfigurasjonsfilen vår. Vi finner ut hvilken tretype som skal knyttes til treposisjonen *pos* ved å velge et tilfeldig tall r mellom 0 og $(T_N - 1)$. Grunnen til dette er at alle tretypene gitt i skogkonfigurasjonsfilen blir puttet inn i en vektor (der indeksen til den første tretypen i vektoren begynner på 0 og siste tretypen i vektoren har indeks $(T_N - 1)$). Vi kommer tilbake til dette i seksjon 5.3.

Vi knytter dermed alle utplasserte treposisjoner i en tile t med en av tretypene som er gitt i skogkonfigurasjonsfilen. For å undersøke om algoritme 4.13 gir ønsket resultat ser vi på en kjøring med for eksempel *spread* variablene som i andre kolonne i tabellen under. Fjerde kolonne i tabellen gir hvor mange prosent av de totale utplasserte treposisjonene for rasterceller med henholdsvis bar- og løvskog som terrengidentifikasjon.

Tretype T	$T.type$	$T.Spread$	Prosent av antall utplasserte trær
M_TREE4	evergreen	20	20.8532%
M_TREE5	evergreen	40	39.5464%
M_TREE6	evergreen	40	39.6005%
P_WILLOW	evergreen	30	32.8598%
M_TREE1	evergreen	70	67.1402%

Som vi ser av tabellen over, er det liten forskjell på *spread* verdien til et tre og antallet utplasserte trær av denne tretypen i landskapet. Vi vil aldri oppnå helt nøyaktig mål på grunn av at vi arbeider med vilkårlig valgte tall men resultatet er mer enn godt nok for vårt formål. Figur 4.15 viser resultatet av trefordelingen med verdiene til tabellen over.



Figur 4.15: *Et eksempel på hvordan forskjellige tretyper kan bli utplassert i et landskap gitt spread verdiene som i tabellen over. Rasterdatasettet som vi arbeider med i denne oppgaven har 325 løvskogceller (≈ 0.495911% av alle celler i rasteret), 37212 barskogceller (≈ 56.781%) og 859 blandet skog celler (≈ 1.31073%). Varianter av lilla i figuren over er treposisjoner som har tretype barskog, og varianter av lyseblå er treposisjoner med tretype løvskog.*

4.6 Dynamisk minnehåndtering

Vi ønsker at skogmodellen skal kunne brukes i samarbeid med andre modeller i større landskapsvisualiseringer. For eksempel kan det tenkes at vi ønsker å implementere skogmodellen sammen med andre hierarkiske modeller av vann eller terreng. Typisk vil skogmodellen være en liten del av en langt større visualiseringsapplikasjon og vi må dermed sørge for at modellen minimerer

bruken av tilgjengelige maskinressurser. Vi konsentrerer oss om å minimere bruken av maskinminne.

Vi er da særlig interessert i å finne løsninger som sørger for å minimere minneforbruket dynamisk. Med dynamisk mener vi at denne prosessen skal foregå under kjøringen av applikasjonen. Til nå har vi generert hele skogmodellen i en preprosess. Dette vil si at etter preprosesseringen er ferdig eksisterer hele skogmodellen i maskinminnet. Dette er ikke ønskelig siden det vil alltid være tiler i quadtreet som ikke er synlige under visualiseringsprosessen. Vi søker i stedet en metode for å instansiere tiler etter behov.

4.6.1 Tiler etter behov

Vi starter med å generere rotnoden i quadtreet samt tiler ned til et gitt nivå. Dette nivået, l_M, settes i skogkonfigurasjonsfilen og bestemmer hvor mange tiler i quadtreet som skal instansieres under preprosesseringen av skogmodellen. Alle tiler fra rotnoden og ned til nivå $l_t = l_M$ instansieres på vanlig måte som nevnt før. Vi ønsker at tiler fra rotnoden og ned til $l_t = l_M$ alltid skal eksistere i maskinminnet.

Vi må også innføre en lokal teller τ_t i hver tile t som sier noe om når t sist ble traversert. For å sette τ_t innfører vi *curr_frame* som en global teller som økes for hver bilderamme som tegnes til skjerm. Ved å sette $\tau_t = curr\_frame$ for hver gang vi traverserer t, vet vi at hvis:

$$\tau_t < \text{curr\_frame}, \tag{4.26}$$

er tilfredsstilt, har vi ikke traversert t under opptegningen av denne bilderammen. Dette vil enten si at t sin omringede sfære B_t ligger utenfor synsvolumet, eller vi har flyttet det virtuelle kameraet lengre unna landskapet slik at vi ikke lenger skal traversere ned til t. Den nye traverseringsalgoritmen er gitt ved følgende algoritme:

Algoritme 4.14, traversere(t) [III]

```
1  beregn ε_t
2  hvis B_t ikke befinner seg innenfor synsvolumet
3     avslutt traverseringen for t
4  τ_t = curr_frame
5  hvis μ_t ≠ 0
6     hvis δ_t > ε_t og l_t < l_maks
7        for alle barn b til t
8           hvis b ikke er instansiert
9              instansier b
10          traversere( b )
```

Ved å sette τ_t etter synlighetssjekken mot B_t i algoritme 4.14 unngår vi å sette τ_t i en tile t som ikke er synlig på skjermen. Vi legger også merke til

at vi for en hver tile t må spørre om $l_t < l_{maks}$ i stedet for $l_t \leq l_{maks}$ som før. Grunnen til dette er at om vi ikke hadde sjekket om $l_t < l_{maks}$ ville vi ha instansiert barn for tiler på det maksimale nivået l_{maks} i quadtreet.

Før vi traverserer videre ned i barna til t må vi undersøke om de allerede er instansiert. Viser det seg at barna til t ikke er instansiert, sørger algoritme 4.14 for å gjøre dette før vi traverserer ned i dem. Dermed er vi sikre på at quadtretraverseringen ikke feiler ved å traversere ned i en tile som ikke finnes.

Vi har foreløpig ikke sett på problemet med minneforbruket til skogmodellen. Det første vi gjør for å løse dette er å innføre et mål på størrelsen som en tile opptar i maskinminnet. Dette målet må være avhengig av hvor mange treposisjoner som er utplassert i tilen. La M_{pos} være antall bytes som en instans av et TrePosisjons objekt opptar i minnet. Gitt antall utplasserte trær i en tile t (μ_t) kan vi dermed si at minneforbuket M_t til t er:

$$M_t = \mu_t M_{pos}. \tag{4.27}$$

Vi innfører M_{maks} som det maksimale minneforbruket til skogmodellen. Verdien til M_{maks} gis i skogkonfigurasjonsfilen. For å holde orden på hvor mye maskinminne skogmodellen bruker til enhver tid innfører vi en global teller M_{curr}. Hver gang vi instansierer en ny tile t må vi sørge for at M_{curr} oppdateres ved:

$$M_{curr} = M_{curr} + M_t. \tag{4.28}$$

Etter endt traversering av quadtreet må vi for hver bilderamme undersøke om:

$$M_{curr} < M_{maks}. \tag{4.29}$$

Hvis $M_{curr} \geq M_{maks}$ må vi rydde opp i quadtreet siden den øvre minnetoleransen M_{maks} er overskredet. Opprydningsprosessen går ut på å søke igjennom quadtreet til vi finner en tile t hvor ligning (4.26) er tilfredsstilt. Denne kan dermed slettes fra quadtreet siden den ikke har vært traversert for denne bilderammen. Vi kan også slette subtreet til t fordi vi vet at disse heller ikke har vært traversert dersom t ikke har vært traversert. Opprydningsalgoritmen gis som:

Algoritme 4.15, ryddOpp(t)

```
1 for alle barn b til t
2   hvis τ_b < curr_frame og l_b > l_M
3     slett subtreet til b
4     slett b
5   ellers
6     ryddOpp( b )
```

Siden vi ønsker at alle tiler t fra rotnoden og ned til nivå $l_t = l_M$ ikke skal slettes fra quadtreet, må vi undersøke om $l_t > l_M$ før vi eventuelt sletter t. For hver tile t som slettes må vi huske på å oppdatere M_{curr} ved å trekke M_t fra M_{curr}. Algoritme 4.15 vil gå rekursivt igjennom hele quadtreet og slette alle tiler som ikke bidro til den opptegnede bilderammen. Valget av M_{maks} vil kontrollere hvor ofte algoritme 4.15 blir kalt.

For valg av stor M_{maks} vil opprydningsalgoritmen bli kalt etter at mange tiles er instansiert. I verste tilfelle kan M_{maks} være så stor at algoritme 4.15 aldri blir kalt. Velger vi derimot en lav M_{maks} kan vi risikere at vi alltid kaller algoritme 4.15 fordi ligning (4.29) alltid vil være tilfredsstilt. Siden et godt valg av M_{maks} vil varierere avhengig av hvor mange trær man har utplassert i landskapet (som igjen er avhengig av den brukerdefinerte η), velger vi å la brukeren selv bestemme verdien for denne i skogkonfigurasjonsfilen. Dette gjøres også for å ta hensyn til ulik maskinvare siden vi for en datamaskin med stor minnekapasitet kan tillate at skogmodellen bruker mer av de tilgjengelige maskinressursene enn for en datamaskin med begrenset minne.

Til slutt må vi nevne at skogmodellen nødvendgis "halter" litt under opprydningsprosessen. Særlig er dette merkbart for høye verdier av M_{maks} som fører til at et stort antall tiler må slettes når algoritme 4.15 først blir kalt. Vi kan forhindre mye av dette ved å senke M_{maks} slik at opprydningen forekommer oftere.

4.6.2 Tileidentifikasjon

Når vi sletter en tile t i algoritme 4.15 og instansierer den igjen ved en senere anledning, noe som kan forekomme dersom man for eksempel flyr over en lang strekning og vender tilbake igjen, ønsker vi at alle de utplasserte trærne i t skal havne på samme sted. Med andre ord: vi ønsker at en tile alltid skal ha samme utseende dersom den må instansieres på nytt under kjøringen av applikasjonen.

Vi vet at tilfeldige tall i en datamaskinen genereres ved å gi et frø til maskinens tilfeldighetsgenerator. Resultatet av dette er en lang sekvens med tilfeldige tall. Samme frø fører til samme rekke med tall [13, side 275].

Vi begynner med å innføre en unik tileidentifikasjon t_{ID} for hver tile t. Vi setter t_{ID} som frøet til t før vi begynner med å utplassere treposisjoner i t. Siden vi finner treposisjonene til t ved å bruke vilkårlige tall, vil vi få de samme treposisjonene dersom vi sørger for å gi samme frø (samme t_{ID}) før vi genererer sekvensen med tilfeldige tall for hver tile. Dette gjøres ved å utnytte orienteringen o_t til t i forhold til foreldretilen.

Vi starter med å sette en unik tileidentifikasjon t_{ID} for rotnoden. Det finnes en mengde metoder for å finne et unikt tall som vil variere fra hver gang man starter opp en applikasjon. I denne oppgaven har vi valgt t_{ID} for rotnoden ved å se på tiden applikasjonen ble startet på. Dette sørger for at

genereringen av treposisjoner i skogmodellen blir unik for hver kjøring.

Etter å ha funnet t_{ID} for rotnoden i quadtreet må vi sørge for at alle tiler nedover i quadtreet får en unik tileidentifikasjon. For at dette skal virke må t_{ID} nødvendigvis være avhengig av foreldretilen sin t_{ID}. Gitt foreldretilen p til en tile t, samt orienteringen til t i forhold til p, kan vi sette t_{ID} for t ved:

$$t_{ID} = 4(p_{ID} + 1) + o_t, \tag{4.30}$$

Siden et quadtre har fire barn, vil ligning (4.30) gi et unikt tall for hver av de fire barna til en tile. Dette skyldes at orienteringen o_t er forskjellig for hvert barn. Vi behandler her orienteringen o_t som om det var et heltall.

Kapittel 5

Trerepresentasjon

Vi husker fra seksjon 1.3.2 at vi skiller mellom treposisjonene i quadtreet og de individuelle trerepresentasjonene som brukes for å tegne et individuelt tre til skjermen. I dette kapittelet skal vi kun konsentrere oss om hvordan vi representerer de ulike trærne i skogmodellen. Som nevnt i seksjon 3.2.1 består de individuelle tremodellene av ulike "Level-of-Detail" representasjoner som genereres i en preprosess.

Når vi snakker om "Level-of-Detail" metoder i denne oppgaven er det viktig å skille mellom LOD representasjonen for quadtreet og LOD representasjonene for trær. Quadtreet blir traversert som en synsavhengig LOD modell som nevnt i seksjon 4.3. Når vi er nærme nok en tile i quadtreet til at vi ønsker å tegne individuelle trær, bruker vi LOD representasjonene til tretypene for å velge hvor detaljerte alle trærne i en tile skal være.

Vi begynner med å introdusere de ulike LOD representasjonene som brukes for å representere et tre i denne oppgaven. Disse blir så knyttet opp mot en diskret LOD modell. Vi ser så på hvordan informasjonen om hvert tre er organisert i skogmodellen, før vi til slutt spesifiserer de ulike LOD representasjonene for en gitt treposisjon i landskapet.

5.1 Trerepresentasjoner

En skog består av mange tusen trær. Polygonrepresentasjoner av trær er som oftest svært kompliserte med hensyn på antall grafiske primitiver. En scene med polygonrepresentasjoner for trær og planter kan fort overstige flere milliarder grafiske primitiver. Dette gjør dem ubrukelige for sanntidsvisualisering [14].

For å sørge for rask visualisering av flere tusen trær søker vi enkle trerepresentasjoner som består av få grafiske primitiver. Trerepresentasjonene vi bruker i denne oppgaven er svært enkle, men gir et godt resultat så lenge man ikke befinner seg altfor nært et tre. De ulike representasjonene er introdusert under.

5.1.1 Planrepresentasjon

Den første trerepresentasjonen vi introduserer er også den enkleste i den forstand at den består av færrest grafiske primitiver. I datagrafikk er "billboarding" en populær teknikk som går ut på å representere et tredimensjonalt objekt som et todimensjonalt plan. I stedet for å tegne et geometrisk komplisert objekt bestående av flere tusen grafiske primitiver, bruker man en bilderepresentasjon av objektet som "limes" oppå et plan. Dette roteres for hver bilderamme slik at normalen alltid peker mot synsretningen [15, side 225].

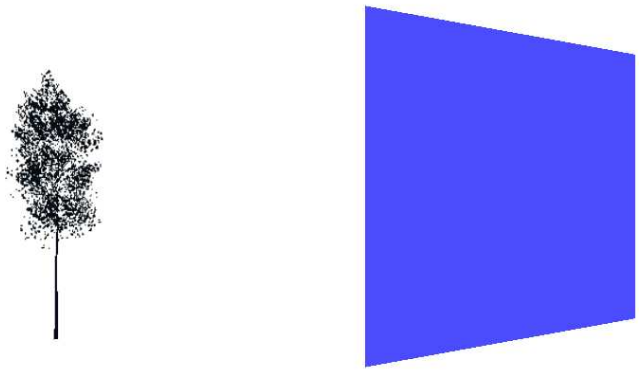
Vi utnytter de samme teknikkene som for "billboarding", men vi ser bort ifra kravet om at normalen alltid må peke mot synsretningen. Vår planrepresentasjon består dermed kun av et plan og en tilhørende tekstur. Akkurat som en rasterbasert dataskjerm består av mange små piksler består en tekstur av mange små teksler. Hver teksel i en tekstur har sin egen tilhørende fargeverdi. I tillegg kan vi velge å assosiere en alfaverdi til en teksel på samme måte som vi gir alfaverdien til en piksel.

Figur 5.1: *Et eksempel på en enkel tretekstur*

Figur 5.1 viser en enkel tekstur av et tre. Ved å sette de hvite tekslene i teksturen på figur 5.1 til å ha alfaverdi 0 vil disse bli gjennomsiktige. Alle andre teksler settes til å ha alfaverdi 1 som betyr at de er helt opake. Vi vil dermed kunne se igjennom de delene av teksturen som ikke tilhører treet. Vi kaller denne teksturen for treteksturen. Et eksempel på en planrepresentasjon av et tre med tilhørende tretekstur er vist på figur 5.2.

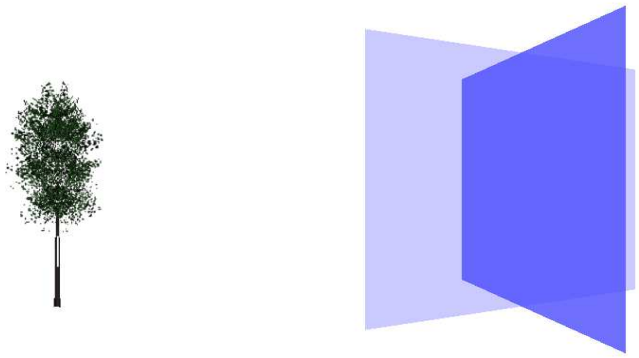
5.1.2 Kryss-planrepresentasjon

Den neste trerepresentasjonen i denne oppgaven har vi valgt å kalle for kryss-planrepresentasjonen. Vi søker en mer tredimensjonal representasjon av objektet og løser dette ved å legge til et plan på tvers av planrepresentasjonen. Kryss-planrepresentasjonen består dermed av to plan som danner



Figur 5.2: *Planrepresentasjonen for et tre. Til venstre ser man tretekstur limt oppå et plan (markert med blått til høyre på figuren).*

et "kryss"-mønster i rommet. Vi bruker samme tretekstur for begge plan. Et eksempel på en kryss-planrepresentasjon er vist på figur 5.3.

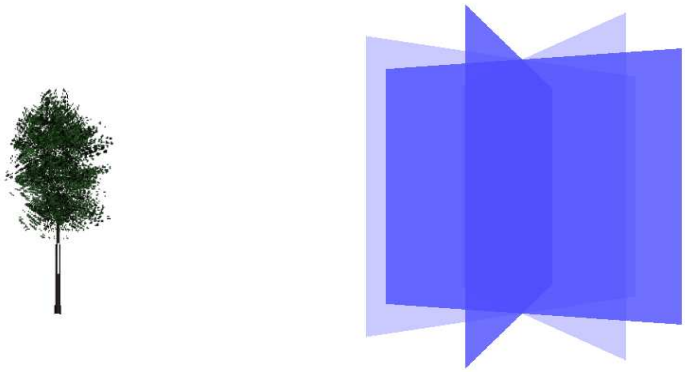


Figur 5.3: *Et eksempel på kryss-planrepresentasjonen. Vi har nå to vinkelrette plan (til høyre) som begge bruker samme tretekstur (til venstre).*

5.1.3 Stjerne-planrepresentasjon

Den siste trerepresentasjonen vi har brukt i denne oppgaven øker den tredimensjonale romfølelsen ved å tilføre to nye plan. Vi danner nå et stjerneformet mønster ved å legge to vinkelrette plan diagonalt mellom kryss-planrepresentasjonen. Stjerne-planrepresentasjonen består dermed av fire plan som vist på figur 5.4.

Vi innfører bladteksturen til et tre. Som navnet tilsier består denne teksturen kun av bladene til treet. Sidene bladene er spredt av natur vil



Figur 5.4: *Stjerne-planrepresentasjonen. To nye vinkelrette plan blir lagt midt mellom kryss-planrepresentasjonen for å danne et stjernemønster i rommet. Til venstre har vi teksturert disse med bladteksturen på figur 5.5. Høyre viser de fire plan som stjerne-planrepresentasjonen består av.*

det gi et bedre visuellt resultat om vi teksturerer noen plan i våre trerepresentasjoner med denne teksturen. Vi lar våre to nye vinkelrette plan som danner stjernemønsteret være teksturert med bladteksturen for et tre. Figur 5.5 viser et eksempel på en bladtekstur for treet på figur 5.1. Som før lar vi de hvite tekslene i bildet ha alfaverdi 0.

Figur 5.5: *Et eksempel på en tekstur som kun inneholder bladene til et tre.*

5.2 Diskret LOD modell for trær

Vi skal nå se på hvordan vi kan knytte trerepresentasjonene introdusert i seksjon 5.1 til en diskret LOD modell. Fra seksjon 3.2 vet vi at en diskret LOD modell består av en mengde pregenererte representasjoner av et objekt. Passende representasjon velges til enhver tid av seleksjonsmekanismen til LOD modellen under kjøringen av applikasjonen. For å beskrive hvordan vår diskret LOD modell er bygget opp må vi ta for oss de tre delene som et LOD rammeverk består av. Disse er generering, seleksjon og skift.

5.2.1 Generering av teksturer

Trerepresentasjoner benytter seg av to teksturer:

- *Tretekstsuren*: Tekstur for å representere hele treet med blader, stamme og greiner.
- *Bladteksturen*: Tekstur som kun består av bladene for et tre.

Hvordan vi skaffer de nødvendige teksturene kan variere. En mulig metode er å fotografere et tre med et kamera. Etter at bildet er overført til datamaskinen må man redigere det ved hjelp av bildebehandligsverktøy.

Det første vi må gjøre for å lage en passende tretekstur er å manuelt fjerne alle detaljer rundt treet slik at bare treet står igjen. Vi må altså erstatte alle tekslene i det originale bildet som ikke tilhører treet med hvitt. Dette gjøres fordi skogmodellen vår forbinder alle hvite teksler i en tretekstur med en alfaverdi på 0. Dermed er det kun treet som er synlig når vi limer denne teksturen på et plan. Jobben med å erstatte alle tekslene i et bilde er en tidkrevende prosess. Gode bildebehandlingsverktøy gjør prosessen enklere men vi må alltid gjøre en god del av jobben selv.

Å generere bladteksturen krever enda mer jobb fordi vi i tillegg manuelt må erstatte alle teksler som representerer trestammen og greiner i teksturen med hvitt. Bladteksturen bruker altså samme utgangspunkt som treteksturen men inneholder enda flere hvite teksler. Figur 5.5 består av samme bilde som figur 5.1 men alle teksler som representerer noe annet enn bladene på treet er erstattet med hvitt.

Kvaliteten på det fotograferte bildet vil da være avgjørende for hvordan treet ser ut i skogsmodellen. Faktorer som tidspunkt på dagen bildet ble tatt, årstid og oppløsning på scanner/bildeformatet vil alle ha innvirkning på hvordan treet ser ut. Det er dermed vanskelig å garantere at det ferdige bildet gir et godt resultat.

En annen fremgangsmåte for å skaffe gode teksturer er å bruke polygonbaserte tremodeller. Programmer som xfrog [14] genererer tremodeller som består av flere tusen grafiske primitiver. De fleste polygonbaserte tremodellene bruker teksturer fra fotografier som legges oppå polygonene for å få en fotorealistisk trerepresentasjon av svært høy kvalitet.

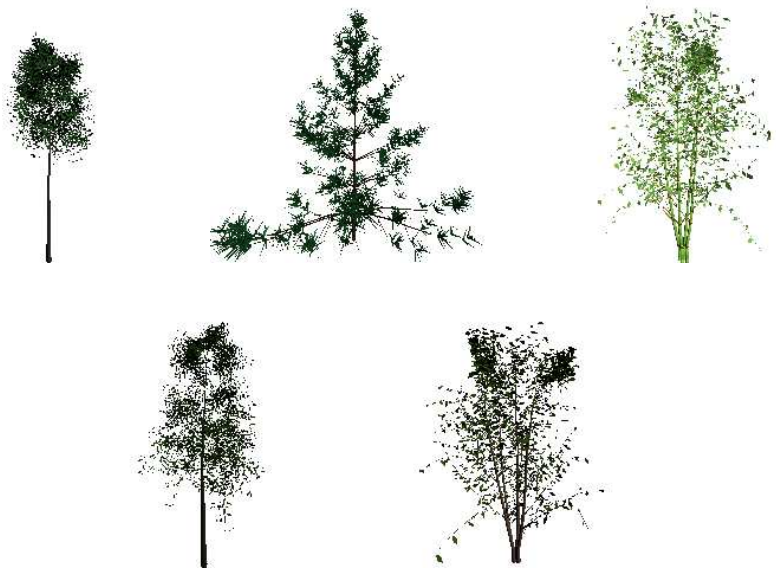
Fordelene med å bruke en polygonbasert modell er:

- *Enkelt å generere teksturer*: En polygonbasert trerepresentasjon gjør at vi slipper å redigere det digitale bildet for hånd. Vi begynner med å laste polygonmodellen inn i et visualiseringsprogram som kan lese filformatet som modellen er i. Slike program gjør det enkelt å spesifisere en bakgrunnsfarge til vinduet. Vi setter denne til hvitt før vi roterer og plasserer tremodellen der vi vil ha den i vinduet. Til slutt lagrer vi resultatet som et bilde. Siden nesten alle polygonbaserte modeller skiller mellom polygonene som utgjør bladene og trestammen (fordi disse

har ulike egenskaper i forhold til refleksjon av lys), er det lett å lage bladteksturen til treet. Man sier bare fra til visualiseringsprogrammet at man ikke ikke ønsker å tegne polygonene som utgjør trestammen. Resultatet er at kun bladene tegnes til skjerm. Vi lagrer så dette bildet som bladteksturen til treet.

- *Lett å endre på modellen*: Polygonbaserte trerepresentasjoner gjør det også lett å endre på parametre som lyssetting for å få bedre tilpasset bilder for en visualisering. Noen programpakker gjør det også mulig å endre på selve modellen ved for eksempel å tilføre flere greiner eller blader. Dette gjør det enkelt å lage flere representasjoner av samme tre.

I denne oppgaven har vi brukt polygonbasert tremodeller. De resulterende tretekstureneene for disse modellene er vist på figur 5.6.



Figur 5.6: *De ulike tretekstureneene brukt i denne oppgaven.*

Før vi går videre må vi innføre to krav til våre teksturer:

- Trestammen må være sentrert i midten av teksturen. Er ikke dette kravet tilfredsstilt vil trepresentasjonene i seksjon 5.1 gi et dårlig visuelt resultat. Dette kommer av at vi bruker flere plan som krysser hverandre. Disse planene krysser hverandre i midten av hvert plan. Hvis ikke trestammen befinner seg midt på teksturen vil trerepresentasjonene gi kunstige effekter.

- Vi ønsker at alle plan som våre trepresentasjonene består av skal være kvadratiske. For å forhindre at de tilhørende teksturene som limes oppå dem blir strukket i en eller annen retning, må også teksturene være kvadratiske.

5.2.2 Seleksjon av trerepresentasjon

La oss nå anta at vi har tilgang på teksturene vi bruker for trerepresentasjonene våre. Neste skritt for å implementere en "Level-of-Detail" modell er å vite når vi skal skifte fra en LOD representasjon til en annen. Før vi gjør dette må vi spesifisere hvordan vi ordner våre ulike trerepresentasjoner i et LOD hierarki.

Vi har de tre ulike trerepresentasjonene som vi presenterte i seksjon 5.1. Vi knytter disse opp i et LOD hiearki ved å si at planrepresentasjonen er det laveste nivået (nivå 0) i vårt LOD hierarki. Husk at vi med det laveste nivået i LOD sammenheng mener den groveste representasjonen av det originale objektet. Det midterste nivået (nivå 1) i vårt LOD hierarki er kryss-planrepresentasjonen som gir en finere representasjon av treet. Øverste nivået i hierarkiet (nivå 2) blir dermed stjerne-planrepresentasjonen som gir den mest detaljerte representasjonen av et tre i skogmodellen.

Vi nevnte i seksjon 3.2 at en "Level-of-Detail" modell bør være slik at antallet grafiske primitiver fordobles for hver modell oppover i hierarkiet. Siden et plan kan dekomponeres til 2 trekanter er antall grafiske primitiver for planrepresentasjonen dermed 2. Kryss-planrepresentasjonen tilfører enda et plan så antall trekanter for denne trerepresentasjonen er 4. Til slutt består stjerne-planrepresentasjonen av 4 plan; altså 8 trekanter. Vi får da følgende:

LOD Nivå	Representasjon	Antall trekanter
2	stjerne-planrepresentasjon	8
1	kryss-planrepresentasjon	4
0	planrepresentasjon	2

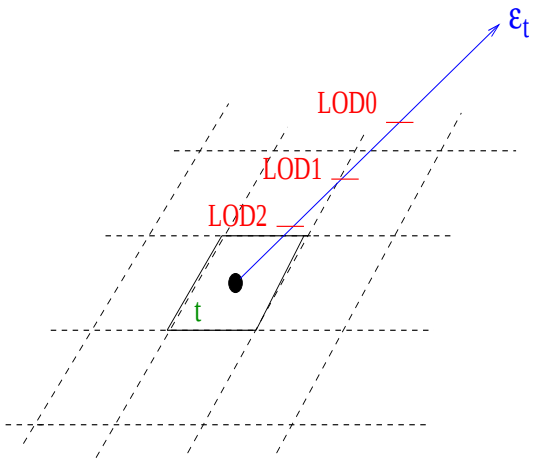
Nå som vi har knyttet våre trerepresentasjoner til et LOD hierarki, er det opp til seleksjonsmekanismen å finne ut når vi skal tegne de ulike trerepresentasjonene. I skogkonfigurasjonsfilen er vi gitt tre parametre på formen:

LOD_0 : <tall>
LOD_1 : <tall>
LOD_2 : <tall>

Vi innfører $LOD_i, i = 0, 1, 2$ for å representere de ulike verdiene. Hensikten med disse er å bestemme når vi skal tegne den i'te representasjonen i vårt LOD hierarki. For å hjelpe oss med dette minner vi om vårt avstandsmål ϵ_t som målte avstandsen fra en tile t i quadtreet til kameraet. Vi ønsker å

bruke ϵ_t til å også velge hvilken LOD representasjon som skal velges for alle trærne t.

LOD_0 gir ϵ_t-verdien som gjør at vi skal tegne planrepresentasjonene i en tile t. Tilsvarende gir LOD_1 ϵ_t-verdien for å tegne kryss-planrepresentasjonene i t og LOD_2 gir når vi skal velge stjerne-planrepresentasjonene. Dette kan illustreres ved figur 5.7.



Figur 5.7: *Figuren viser hvordan vi knytter de ulike LOD_i grensene til avstandsmålet ϵ_t for en tile t.*

For at seleksjonsmekanismen skal være riktig må vi kreve at:

$$LOD_0 > LOD_1 > LOD_2 \tag{5.1}$$

Gitt LOD_i og ϵ_t er vi nå klare for å gi algoritmen som sørger for å velge riktig LOD representasjon for en tile t:

Algoritme 5.1, finnLODRepresentasjon(ϵ_t)

```
1 hvis εt > LOD0
2     ikke velg LOD representasjon
3 ellers hvis LOD1 < εt ≤ LOD0
4     velg LOD0
5 ellers hvis LOD2 < εt ≤ LOD1
6     velg LOD1
7 ellers hvis εt ≤ LOD2
8     velg LOD2
```

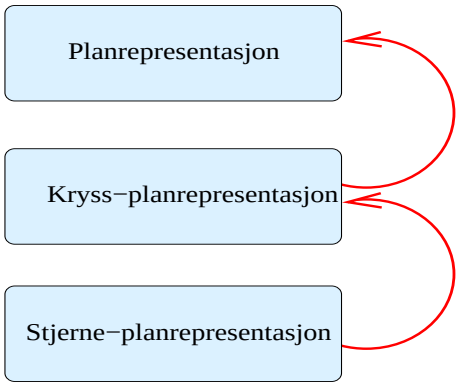
Alle trær i en tile t blir tegnet med samme detaljgrad. Så lenge $\epsilon_t > LOD_0$ skal vi ikke velge noen LOD representasjon. Dette tilsvarer at vi ikke skal tegne trær i en tile når avstanden fra tilen er for stor. Dermed har vi tatt

hensyn til antagelsen i seksjon 1.3.1 om at vi ikke klarer å skille individuelle trær fra hverandre hvis vi befinner oss alt for langt unna landskapet.

5.2.3 Skift mellom ulike trerepresentasjoner

Vi ønsker at skiftene mellom våre ulike "Level-of-Detail" representasjoner skal være så lite synlig som mulig. For å gjøre dette bruker vi alfablending. Metoden som forklares her er en variant av den generelle fremgangsmåten for "Blend LODs" introdusert i seksjon 3.2.1.

Går vi tilbake til kryss-planrepresentasjonen, kan vi tenke på denne som bestående av to separate deler: den opprinnelige planrepresentasjonen og et plan som står vinkelrett på denne. Vi kan derfor forenkle kryss-planrepresentasjonen ved å si at denne kun består av ett plan som skal tegnes oppå planrepresentasjonen. Når vi tegner kryss-planrepresentasjonen må vi sørge for å samtidig tegne planrepresentasjonen. Tilsvarende kan vi si at en stjerne-planrepresentasjonen kun består av to plan. For å tegne stjerne-planrepresentasjonen må vi altså samtidig tegne kryss-planrepresentasjonen (som igjen sørger for å tegne den opprinnelige planrepresentasjonen). Relasjonene mellom disse er vist på figur 5.8.



Figur 5.8: *Relasjonene mellom våre ulike trerepresentasjoner. Stjerne-planrepresentasjonen er avhengig av kryss-planrepresentasjonen som igjen er avhengig av planrepresentasjonen.*

Når vi skal bytte fra planrepresentasjonen til kryss-planrepresentasjonen holder det altså med å kun blende inn ett plan. Tilsvarende kan vi si at det holder med å blende inn to plan når vi skal bytte fra kryss- til stjerne-planrepresentasjonen. Fordelen med dette er at vi aldri behøver å blende ut en LOD modell samtidig som vi blender inn en ny modell.

Vi må ha et mål på over hvor lang avstand blendingen skal foregå. Dette vil si at vi må ha gitt en blendinglengde. Denne er gitt i skogkonfigurasjons-filen på formen:

```
transition : <tall>
```

Blendinglengden kaller vi for *trans*.

Gitt de ulike LOD_i grensene og en *trans* verdi er vi klare for å introdusere blendingprosessen mellom våre trerepresentasjoner. Vi introduserer α_{LOD_0} for alfaverdien for planrepresentasjonen, α_{LOD_1} for alfaverdien til kryss-planrepresentasjonen og α_{LOD_2} for alfaverdien til stjerne-planrepresentasjonen. Vi søker en funksjon $f_i(\epsilon_t)$ som, gitt LOD_i, er slik at:

$$f_i(\epsilon_t) = \begin{cases} 0, & \epsilon_t = LOD_i, \quad \text{i} = 0, 1, 2 \\ 1, & \epsilon_t = LOD_i - trans, \quad \text{i} = 0, 1, 2, \end{cases} \tag{5.2}$$

$f_i(\epsilon_t)$ må dermed være på formen:

$$f_i(\epsilon_t) = \frac{LOD_i - \epsilon_t}{trans} \tag{5.3}$$

Vi setter de ulike alfaverdiene til LOD representasjonene våre ved:

$$\alpha_{LOD_0} = f_0(\epsilon_t) \tag{5.4}$$
$$\alpha_{LOD_1} = f_1(\epsilon_t) \tag{5.5}$$
$$\alpha_{LOD_2} = f_2(\epsilon_t) \tag{5.6}$$

Når $\epsilon_t = LOD_0$ skal vi begynne med å blende inn planrepresentasjonene innenfor tilen t. Dette vil si at $\alpha_{LOD_0} = 0$ når $\epsilon_t = LOD_0$. Etterhvert som vi nærmer oss t, som svarer til at avstanden ϵ_t til t blir mindre, skal vi øke α_{LOD_0} helt til $\epsilon_t = LOD_0 - trans$. Her ønsker vi at $\alpha_{LOD_0} = 1$.

Ved å sette $\alpha_{LOD_0} = f_0(\epsilon_t)$ som i ligning (5.3) vil dette være oppfylt. Vi ser at:

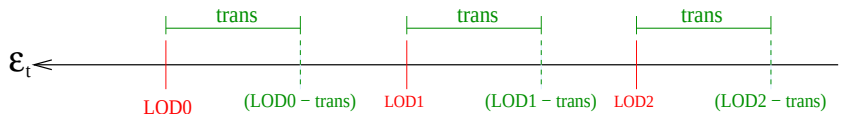
$$f_0(LOD_0) = \frac{LOD_0 - LOD_0}{trans} = 0, \tag{5.7}$$

og

$$f_0(LOD_0 - trans) = \frac{LOD_0 - (LOD_0 - trans)}{trans} = \frac{trans}{trans} = 1. \tag{5.8}$$

Synkende verdier for ϵ_t fra LOD_0 til $LOD_0 - trans$ vil gi økende $f_0(\epsilon_t)$. Vi ser dermed $\alpha_{LOD_0} = f_0(\epsilon_t)$ gir ønsket blending. Tilsvarende vil $\alpha_{LOD_1} = f_1(\epsilon_t)$ for kryss-planrepresentasjonen og $\alpha_{LOD_2} = f_2(\epsilon_t)$ for stjerne-planrepresentasjonene gi ønsket blendingresultat. Vi kaller $f_i(\epsilon_t)$ i ligning (5.3) for en blendingfunksjon.

Figur 5.9 illustrerer hvordan vi skal blende inn de ulike LOD representasjonene i forhold til ϵ_t. Vi blender inn den $i'te$ representasjonen fra $\epsilon_t = LOD_i$ til $\epsilon_t = LOD_i - trans$. Når $\epsilon_t = LOD_i - trans$ er vi ferdig med å blende inn den i'te modellen i LOD hierarkiet. Dette vil si at $\alpha_{LOD_i} = 1$.



Figur 5.9: *Et eksempel på når man skal tegne de ulike LOD_i representasjonene for ϵ_t. Vi gir ϵ_t som stigende fra høyre mot venstre. Dette tilsvarer at avstanden til en tile t blir større etterhvert som vi beveger oss i denne retningen langs ϵ_t-linjen.*

For at blendingen skal gi riktig resultat må vi kreve at:

$$LOD_0 - trans > LOD_1 \tag{5.9}$$

og

$$LOD_1 - trans > LOD_2 \tag{5.10}$$

Uten dette kravet har vi ingen garanti for at en LOD representasjon er ferdig blendet inn i landskapet før vi skal blende inn neste trerepresentasjon. Ved å sørge for at ligning (5.9) og (5.10) er oppfylt vil vi unngå denne problemstillingen.

5.3 Trær i skogmodellen

Vi har til nå både introdusert en "Level-of-Detail" modell som representerer trær på ulike detaljnivå, samt en quadtrestruktur som plasserer ut treposisjoner i landskapet. Vi har også sett hvordan vi knytter opp avstandsmålet ϵ_t til både å traversere quadtreet og velge LOD representasjon for trærne i en tile. Hensikten nå er å ta for oss hvordan ulike tretyper knyttes til en treposisjon. Dette er dermed en videreføring av seksjon 4.5.

5.3.1 Organisering av data

I skogen vår er det mange objekter (trær) med nesten identiske trekk. Utnytter vi ikke sammenhengen mellom like trær i skogmodellen, vil en enhver instans av et *TrePosisjon* objekt være en kostbar operasjon i form av minneforbuk. Vi ønsker å forhindre at alle *TrePosisjon* objekter har sine egne lokale kopier av informasjonen som er felles for samme tretype.

En grunn til at vi ønsker å gjøre dette kommer av minnehåndteringsmekanismen introdusert i seksjon 4.6.1. Minneforbruket M_t til en tile t i

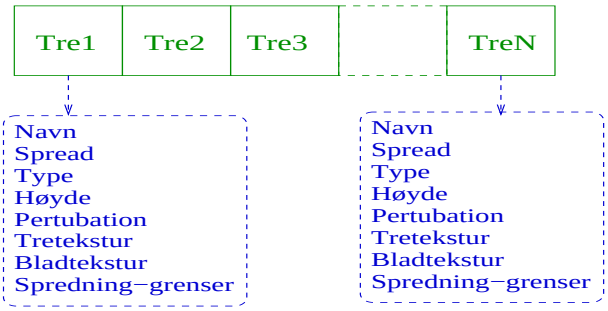
quadtreet vårt er avhengig av antall utplasserte trær μ_t i t og størrelsen (i bytes) til en instans av en treposisjon M_{tre}. Jo mindre M_{tre} er, jo flere trær kan vi utplassere i en tile t før vi overskrider den maksimale minnetoleransen M_{maks} i skogmodellen.

5.3.2 Felles informasjon for en tretype

Etter at applikasjonen har lest inn skogkonfigurasjonsfilen instansieres et *TreType* objekt for alle tretypene gitt i filen. Hvert *TreType* objekt inneholder verdiene til alle variablene som er gitt for en tretype i skogkonfigurasjonsfilen: *name*, *spread*, *type*, *height* og *pertubation*. For en forklaring på betydningen av de ulike parametrene henvises leseren til tillegg A.

I tillegg til dette har hver *TreType* objekt sine egne kopier av tre- og bladteksturene for tretypen. Fra seksjon 4.5 husker vi at vi introduserte *spredning-grenser* som hjelp oss med å finne hvilken tretype som skulle knyttes til et *TrePosisjon* objekt i en tile t. Til slutt må vi altså knytte spredning-grensene til alle *TreType* objekter.

Alle *TreType* objektene blir lagt i en vektor etter at skogkonfigurasjonsfilen er lest inn. Vektoren må være tilgjengelig for hele quadtreet for at det skal være mulig å vite hvilken tretype som skal knyttes til en treposisjon. Figur 5.10 viser denne vektoren etter at vi har lest inn en skogkonfigurasjonsfil, lastet inn teksturene og beregnet spredning-grenser for alle *TreType* objekter i skogmodellen.



Figur 5.10: *Etter at vi har lest skogkonfigurasjonsfilen vår har vi all informasjonen tilgjengelig for å instansiere et TreType objekt. Disse legges inn i en vektor som vist på figuren.*

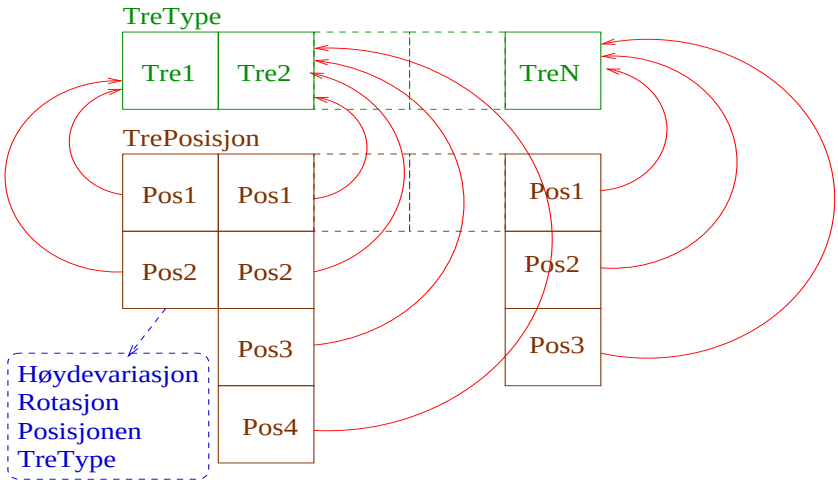
5.3.3 Individuell informasjon for en treposisjon

Vi nevnte i seksjon 1.3.1 at vi kun trenger noen få trerepresentasjoner for å bygge opp skogmodellen. Hvis vi varierer rotasjon og høyde for hvert tre i landskapet vårt vil vi oppnå en god illusjon av en skog. Et *TrePosisjon* objekt inneholder derfor følgende fire variable:

- *Høydevariasjon* : Hvor høyt treet skal være i forhold til tretypens *height* variabel. Vi kaller denne h_Δ.
- *Rotasjon* : Hvor mye treet skal roteres rundt sin egen høydeakse. Vi kaller denne ϕ_Δ.
- *Posisjon* : Koordinatene til treposisjonen i tilen t
- *TreType* : Peker til det tilhørende TreType objektet

For hver tile t innfører vi en lokal vektor som inneholder alle *TrePosisjon* objektene. Hver gang et *TrePosisjon* objekt blir lagt til t i algoritme 4.14 beholder vi indeksen til posisjonen for den tilhørende tretypen i *TreType* vektoren. Vi legger så det nye *TrePosisjon* objektet inn i den lokale *TrePosisjon* vektoren til t ved å gi samme indeks som det tilhørende *TreType* objektet. Dermed blir alle *TrePosisjon* objektene for samme *TreType* tilgjengelig etter hverandre.

Et eksempel på dette er gitt på figur 5.11. Grunnen til at vi organiserer *TrePosisjon* objektene slik er av visualiseringshensyn. Siden hver treposisjon i terrenget med samme tretype nødvendigvis vil bruke samme teksturerer, er det effektivt å tegne alle treposisjonene for en tretype før vi tegner neste tretype.



Figur 5.11: *Et eksempel på hvordan vi for en gitt tile t organiserer TrePosisjon objektene etter indeksen til den tilhørende tretypen. Alle TrePosisjon objekter har mulighet for å få tak i informasjon i TreType objektet gjennom sin lokale TreType peker.*

Alle teksturer som skal brukes av en applikasjon ligger lagret i hovedminnet på datamaskinen og overføres til teksturminnet på grafikkortet når teksturen brukes. Dette teksturminnet er ofte lite men til gjengjeld svært

raskt. En "texture cache miss" skjer når teksturen som grafikkakseleratoren skal bruke ikke ligger i teksturminne på kortet. Dette fører til at grafikkortet må overføre teksturen fra hovedminne på datamaskinen. Dette er en kostbar operasjon fordi minnebåndbredden er ofte flaskehalsen i moderne grafikksystemer [9, side 689-690].

Ved å tegne alle trær som består av samme tekstur etter hverandre, eller med andre ord alle treposisjoner av samme tretype, unngår vi så mye det lar seg gjøre unødige "texture cache misses". Vi er garantert at teksturen ligger i det dedikerte teksturminnet etter at første treposisjon av denne tretypen er tegnet. Dermed kan neste treposisjon for samme tretype tekstureres med samme tekstur uten at noen teksturoverføring finner sted.

Vi bruker også "mipmaps" for de ulike tre- og bladteksturene. Dette er en metode for å ordne flere teksturer av samme objekt i et hierarki. Hver tekstur nedover i hierarkiet har dårligere oppløsning enn nivået over. OpenGL sørger for automatisk å velge hvilken teksturoppløsning som passer best til å representere et objekt ved å se på hvor mange piksler teksturen dekker på skjerm. "Mipmaps" kan dermed tenkes som et slags "Level-of-Detail" metode for teksturer [15, side 379].

Fordelen med å bruke mipmaps er at teksturer med dårligere oppløsning nødvendigvis vil være mindre enn teksturer med finere oppløsning. Siden disse teksturene er mindre vil de også ta mindre plass i teksturminne på grafikkortet. Dermed får vi plass til flere teksturer om vi velger dårligere teksturoppløsning for å representere et objekt som allikevel er langt unna.

Til slutt ser vi på hvordan vi kan finne høydevariasjonen h_Δ og rotasjonen ϕ_Δ for et *TrePosisjon* objekt. Høyden på treet, gitt av *height* variabelen i *TreType* klassen, angir høyden på treet. Rettere sagt: *height* angir minste høyden til et tre fordi vi ønsker å la høyden variere med et vilkårlig tall h_Δ mellom 0 og *perturbation*, hvor *perturbation* er også gitt for hvert *TreType* objekt.

For å illustrere dette tar vi utgangspunkt i M_TREE4 treet gitt i skogkonfigurasjonsfilen i tillegg A. Denne tretypen er gitt følgende *height* og *pertubation* verdier:

```
height : 2.0
perturbation : 0.5
```

For hvert *TrePosisjon* objekt som er av tretype M_TREE4, begynner vi med å finne et tilfeldig tall h_Δ mellom 0 og 0.5. Den totale høyden til M_TREE4 representasjonen for hvert tilhørende *TrePosisjon* objekt blir dermed $2.0 + h_\Delta$. Likedan lar vi rotasjonen være et vilkårlig tall mellom 0 og $\frac{\pi}{2}$ for hvert *TrePosisjon* objekt. Dermed får hvert eneste utplasserte tre i skogmodellen et lite avvik i både høyde og rotasjon.

5.4 LOD representasjonene for en treposisjon

Vi antar nå at vi har gitt et objekt P av TrePosisjon klassen med tilhørende verdier for h_Δ, ϕ_Δ og posisjonen $p = (p_x, p_y, p_z)$. Vi lar h være den minimale høyden til den tilhørende tretypen til P. Vi ønsker nå å spesifisere de ulike plan som våre trerepresentasjoner består av for en gitt posisjon p i rommet.

Grunnen til dette er at vi ønsker å bruke "display lists". En "display list" er en måte å gruppere flere OpenGL instruksjoner på en effektiv måte. Ved å legge mange instruksjoner i en slik liste, kan man kompilere listen en gang og eksekvere den etter behov ved en senere anledning. For statisk geometri som ofte tegnes til skjerm er "display lists" en rask og effektiv struktur [15, side255-276].

Ulempen med en "display list" er at om man først har gitt en liste kan man ikke endre innholdet i den. Grunnen til dette er effektiviseringshensyn: hadde en "display list" vært mulig å modifisere, hadde effektiviteten til en slik liste vært betraktelig redusert. Ved å forhindre at innholdet i en "display list" kan bli endret er det opp til den individuelle OpenGL implementasjonen å sørge for å optimalisere informasjonen lagret i en slik liste. For eksempel kan en OpenGL implementasjon optimalisere data i en "display list" for raskere prosessering eller overføre listen til et eget minne på grafikkortet.

Siden trær i en skog er statiske passer "display list" strukturen godt til vårt formål. Genereringen av en "display list" for en tile t gjøres etter at vi er ferdig med å utplassere trær i t. Siden vårt LOD hierarki for trær består av tre ulike trerepresentasjoner, må hver tile t i quadtreet inneholde tre "display lists". Vi kaller disse for $list_0^t$ ("Display listen" som kun inneholder planrepresentasjonene i en tile t), $list_1^t$ (inneholder kun kryss-planrepresentasjonene i t) og $list_2^t$ (for stjerne-planrepresentasjonene i samme tile).

La oss nå anta at vi har situasjonen som på figur 5.11 etter at vi har forsøkt å utplassere ω_{l_t} trær i en tile t ved algoritme 4.12. Vi ønsker nå å generere de ulike $list_i^t$ for alle de utplasserte trærne i t. Siden vi kun har gitt et punkt for å beskrive et tre må vi først se på hvordan vi spesifiserer de ulike trerepresentasjonene ut i fra dette punktet.

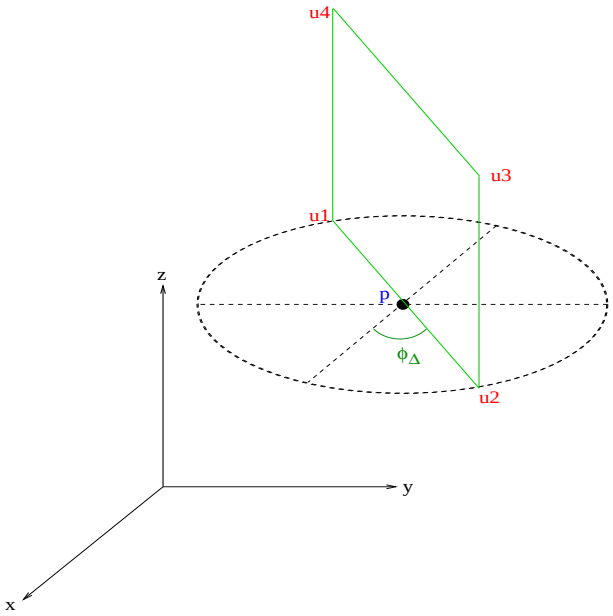
Vi vet at høyden for et TrePosisjon objekt er gitt av $h + h_\Delta$. Dette er midlertidig ikke helt nøyaktig: $h + h_\Delta$ er høyden på de ulike plan som trerepresentasjonene består av. Siden vi søker kvadratiske plan er også bredden $h + h_\Delta$. Gitt en posisjon $p = (p_x, p_y, p_z)$ i rommet, ønsker vi å sentrere våre plan rundt dette punktet. Vi lar dermed p være sentrum i sirkelen med radius:

$$r = \frac{1}{2}(h + h_\Delta). \tag{5.11}$$

Gitt radien til en sirkel som i ligning (5.11), samt et punkt p, er vi nå klare for å spesifisere de ulike trerepresentasjonene for denne treposisjonen.

5.4.1 Spesifisering av planrepresentasjonen

Vi begynner med å spesifisere planrepresentasjonen. Denne består av et plan som vist på figur 5.12. Målet vårt blir dermed å finne de fire punktene (u_1, u_2, u_3, u_4) som planet består av. Som før er p punktet som vi ønsker å sentrere planet rundt og r er radien til sirkelen som i ligning (5.11). I tillegg må vi ta hensyn til ϕ_Δ som er rotasjonen for dette treet rundt z-aksen.



Figur 5.12: *Eksempel på hvordan vi spesifiserer planrepresentasjonen gitt midtpunktet i en sirkel p med radius som i ligning 5.11. Målet er å finne de fire hjørnene* (u_1, u_2, u_3, u_4).

Vi starter med å finne de to hjørnene (u_2, u_3) til planet på figur 5.12. For å ta hensyn til rotasjonen ϕ_Δ er hjørnet u_2 gitt som:

$$(u_{2x}, u_{2y}, u_{2z}) = (p_x + rcos(\phi_\Delta), p_y + rsin(\phi_\Delta), p_z). \tag{5.12}$$

Punktet u_3 har samme (x, y) verdier som u_2 men vi må i tillegg øke z-verdien for å få riktig høyde på planrepresentasjonen. Siden høyden til alle plan i våre trerepresentasjoner er gitt som $h + h_\Delta$, kan vi gi u_3 som:

$$(u_{3x}, u_{3y}, u_{3z}) = (p_x + rcos(\phi_\Delta), p_y + rsin(\phi_\Delta), p_z + (h + h_\Delta)). \tag{5.13}$$

For å finne hjørnet u_1 på figur 5.12 må vi øke rotasjonen ϕ_Δ med π. Vi får da:

$$(u_{1x}, u_{1y}, u_{1z}) = (p_x + rcos(\phi_\Delta + \pi), p_y + rsin(\phi_\Delta + \pi), p_z). \tag{5.14}$$

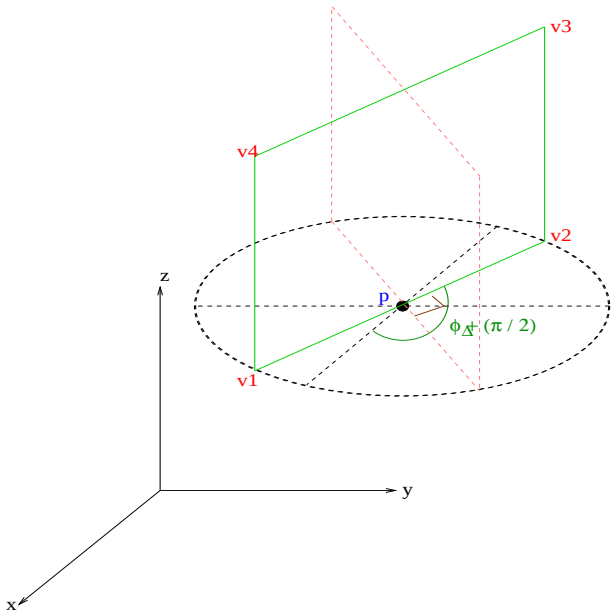
Har vi først funnet (x, y) koordinatene til u_1, finner vi u_4 på samme måte som vi fant u_3 i ligning (5.13):

$$(u_{4x}, u_{4y}, u_{4z}) = (p_x + rcos(\phi_\Delta + \pi), p_y + rsin(\phi_\Delta + \pi), p_z + (h + h_\Delta)). \quad (5.15)$$

Vi har dermed gitt de fire punktene som danner planrepresentasjonen for et punkt p.

5.4.2 Spesifisering av kryss-planrepresentasjonen

Kryss-planrepresentasjonen består av et plan som legges vinkelrett på planrepresentasjonen. Dette planet skal også være sentrert rundt posisjonen p. De fire hjørnene (v_1, v_2, v_3, v_4) som vi nå må finne er gitt på figur 5.13.



Figur 5.13: *Eksempel på de fire hjørnene* (v_1, v_2, v_3, v_4) *som spenner ut kryss-planrepresentasjonen. Figuren viser relasjonen til planrepresentasjonen ved å sørge for at de fire punktene danner et plan som står vinkelrett på denne.*

Vi begynner med å finne hjørnet v_2 til kryss-planrepresentasjonen. Siden vi ønsker et plan som skal stå vinkelrett på planrepresentasjonen, må vi gi rotasjonen rundt p som $\phi_\Delta + \frac{\pi}{2}$. Vi får da:

$$(v_{2x}, v_{2y}, v_{2z}) = (p_x + rcos(\phi_\Delta + \frac{\pi}{2}), p_y + rsin(\phi_\Delta + \frac{\pi}{2}), p_z). \quad (5.16)$$

Tilsvarende er v_3 gitt som samme posisjon som v_2 men med z-verdien økt med $h + h_\Delta$:

$$(v_{3x}, v_{3y}, v_{3z}) = (p_x + rcos(\phi_\Delta + \frac{\pi}{2}), p_y + rsin(\phi_\Delta + \frac{\pi}{2}), p_z + (h + h_\Delta)). \quad (5.17)$$

For å finne hjørnet v_1 må vi gi rotasjonen som:

$$\phi_\Delta + \frac{\pi}{2} + \pi = \phi_\Delta + \frac{3}{2}\pi. \quad (5.18)$$

Vi kan da gi v_1 som:

$$(v_{1x}, v_{1y}, v_{1z}) = (p_x + rcos(\phi_\Delta + \frac{3}{2}\pi), p_y + rsin(\phi_\Delta + \frac{3}{2}\pi), p_z). \quad (5.19)$$

Hjørnet v_4 er gitt som å øke z-verdien til v_1 i ligning (5.19):

$$(v_{4x}, v_{4y}, v_{4z}) = (p_x + rcos(\phi_\Delta + \frac{3}{2}\pi), p_y + rsin(\phi_\Delta + \frac{3}{2}\pi), p_z + (h + h_\Delta)). \quad (5.20)$$

Vi har dermed gitt de fire hjørnene som spenner ut kryss-planrepresentasjonen.

5.4.3 Spesifisering av stjerne-planrepresentasjonen

Stjerne-planrepresentasjonen består av to vinkelrette plan som ligger mellom kryss-planrepresentasjonen. Vi må da finne hjørnene til begge plan som på figur 5.14. Vi gir de fire punktene (s_1, s_2, s_3, s_4) før vi gir (t_1, t_2, t_3, t_4).

For å finne hjørnet s_2 må vi øke rotasjonen ϕ_Δ med:

$$\phi_\Delta + \frac{\pi}{4}. \quad (5.21)$$

Dermed vil s_2 ligge mellom u_2 og v_2 på sirkelen. Vi får da:

$$(s_{2x}, s_{2y}, s_{2z}) = (p_x + rcos(\phi_\Delta + \frac{\pi}{4}), p_y + rsin(\phi_\Delta + \frac{\pi}{4}), p_z). \quad (5.22)$$

Tilsvarende for å finne s_3 er det bare å øke z ved:

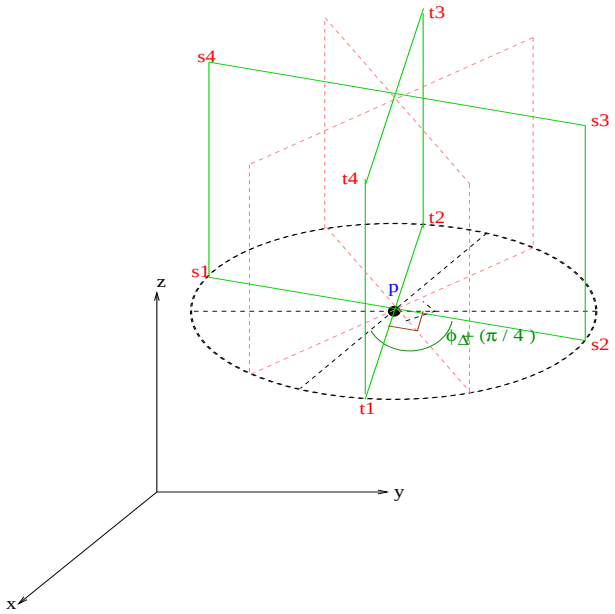
$$(s_{3x}, s_{3y}, s_{3z}) = (p_x + rcos(\phi_\Delta + \frac{\pi}{4}), p_y + rsin(\phi_\Delta + \frac{\pi}{4}), p_z + (h + h_\Delta)). \quad (5.23)$$

For å finne s_1 må vi øke rotasjonen til s_2 med π:

$$\phi_\Delta + \frac{\pi}{4} + \pi = \phi_{Delta} + \frac{5}{4}\pi. \quad (5.24)$$

Vi kan da gi s_1 som:

$$(s_{1x}, s_{1y}, s_{1z}) = (p_x + rcos(\phi_\Delta + \frac{5}{4}\pi), p_y + rsin(\phi_\Delta + \frac{5}{4}\pi), p_z). \quad (5.25)$$



Figur 5.14: *Eksempel på de åtte hjørnene* (s_1, s_2, s_3, s_4) *og* (t_1, t_2, t_3, t_4) *som spenner ut de to plan som danner stjerne-planrepresentasjonen. Figuren viser relasjonen til kryss-planrepresentasjonen.*

Hjørnet s_4 er da gitt ved å øke p_z med $h + h_\Delta$:

$$(s_{4x}, s_{4y}, s_{4z}) = (p_x + rcos(\phi_\Delta + \frac{5}{4}\pi), p_y + rsin(\phi_\Delta + \frac{5}{4}\pi), p_z + (h + h_\Delta)). \tag{5.26}$$

For å spesifisere de fire hjørnene (t_1, t_2, t_3, t_4) på figur 5.14, må vi sørge for at vårt siste plan ligger vinkelrett på planet utspent av (s_1, s_2, s_3, s_4). Vi starter med å finne t_2 ved å øke rotasjonen til s_2 med $\frac{\pi}{2}$:

$$\phi_\Delta + \frac{\pi}{4} + \frac{\pi}{2} = \phi_\Delta + \frac{3}{4}\pi. \tag{5.27}$$

Vi er da klare for å gi hjørnet t_2 som:

$$(t_{2x}, t_{2y}, t_{2z}) = (p_x + rcos(\phi_\Delta + \frac{3}{4}\pi), p_y + rsin(\phi_\Delta + \frac{3}{4}\pi), p_z). \tag{5.28}$$

Vi øker z-verdien til t_2 for å finne t_3 som vanlig:

$$(t_{3x}, t_{3y}, t_{3z}) = (p_x + rcos(\phi_\Delta + \frac{3}{4}\pi), p_y + rsin(\phi_\Delta + \frac{3}{4}\pi), p_z + (h + h_\Delta)). \tag{5.29}$$

Til slutt må vi gi hjørnet t_1 og t_4. Dette gjøres ved å øke rotasjonen til t_2 med π som før:

$$\phi_\Delta + \frac{3}{4}\pi + \pi = \phi_\Delta + \frac{7}{4}\pi \tag{5.30}$$

Dermed er vi klar for å gi hjørnet t_1 som:

$$(t_{1x}, t_{1y}, t_{1z}) = (p_x + rcos(\phi_\Delta + \frac{7}{4}\pi), p_y + rsin(\phi_\Delta + \frac{7}{4}\pi), p_z). \tag{5.31}$$

Som før øker vi p_z for å få hjørnet t_4:

$$(t_{4x}, t_{4y}, t_{4z}) = (p_x + rcos(\phi_\Delta + \frac{7}{4}\pi), p_y + rsin(\phi_\Delta + \frac{7}{4}\pi), p_z + (h + h_\Delta)). \tag{5.32}$$

Vi har da gitt begge plan som stjerne-planrepresentasjonen består av.

5.4.4 Generering av "display lists"

Etter at vi har spesifisert hvordan vi beregner de ulike plan som våre tre-representasjoner består av, skal vi nå se på hvordan vi genererer de ulike $list_i^t$ for en tile t. Vi tar bare for oss genereringen av $list_0^t$, altså listen med planrepresentasjonene i t, da tilsvarende prosess brukes for å generere $list_1^t$ og $list_2^t$. Som nevnt foregår denne prosessen med en gang etter at vi har utplassert trær i en tile t:

Algoritme 5.2, genererList0(t)

```
1  StartList( list_0^t )
2     for alle tilgjengelige TreType objekt T
3        tretekstur = T.tretekstur
4        h = T.hyde
5        for alle TrePosisjon objektene P som tilhrer T
6           h_Δ = P.Hydvariasjon
7           r = ½(h + h_Δ)
8           p = P.posisjon
9           φ_Δ = P.rotasjon
10          Beregn (u_1, u_2, u_3, u_4)
11          Lim tretekstur på planet utspent av (u_1, u_2, u_3, u_4)
```

Her er *StartList(...)* bare en metode for å si ifra at vi nå ønsker å generere en ny "display list".

Vi løper igjennom alle TrePosisjon objektene til hvert TreType objekt og beregner de fire hjørnene (u_1, u_2, u_3, u_4) som danner planrepresentasjonen som i seksjon 5.4.3. Merk at når vi genererer en "display list" er det kun resultatene av beregningene som blir lagret i listen. Dette vil si at når vi ved et senere tidspunkt i applikasjonen vår ønsker å eksekvere $list_0^t$, er allerede

de fire hjørnene (u_1, u_2, u_3, u_4) beregnet og resultatet lagt inn i $list_0^t$. Vi må dermed kun beregne de fire hjørnene til et plan en gang. Dette er en fordel siden trigonometri funksjoner som *cos* og *sin* er dyre med hensyn på prosesseringstid.

Etter at genereringen av alle "display lists" er ferdig, har vi en effektiv metode for å tegne alle trærne for alle detaljrepresentasjoner i t. Vi knytter $list_0^t$ til nivå 0 i LOD hierarkiet, $list_1^t$ til nivå 1 i hierarkiet og $list_2^t$ til nivå 2 i LOD modellen vår. Siden vi forhåndsgenererer alle listene før opptegning er vår LOD modell for trerepresentasjonene en diskret LOD modell.

Kapittel 6

Visualisering

Målet med dette kapittelet er å se på hvordan vi kan visualisere skogmodellen vår. For å gjøre dette må vi kombinere quadtretraverseringen med valg av trerepresentasjoner i hver tile. I denne oppgaven har vi sett på to ulike xmetoder: direkterendering og dybderendering.

Vi begynner med å repetere de ulike parametrene som vi bruker under visualiseringen. Så introdusere vi begge visualiseringsmetodene for det trivielle tilfellet hvor landskapet ligger i et planart område. Deretter ser vi på en forbedret avstandsmål, før vi viser hvordan vi kan tilføre høydeverdier til skogmodellen.

6.1 Repetisjon av begreper

Fra seksjon 4.3 husker vi at vi traverserer quadtreet som en synsavhengig LOD modell. For å gjøre dette innførte vi en objektfeil δ_t for hver tile t i quadtreet. Denne var gitt som:

$$\delta_t = \sigma_t K, \tag{6.1}$$

hvor K var en brukerdefinert konstant og σ_t var størrelsen til tilen t. Siden quadtreet deler en tile i fire like store kvadranter er størrelsen σ_t lik for alle tiler på samme nivå. Dermed kan vi slå fast at også objektfeilen δ_t er lik for alle tiler på samme nivå i quadtreet. For å lette denne diskusjonen innfører vi δ_i, hvor subskriptet i gir oss objektfeilen på nivå i i quadtreet. Rotnoden har dermed objektfeil δ_0, barna til rotnoden objektfeil δ_1 og så videre.

Objektfeilen beregnes for hver tile i quadtreet under instansiering. For å velge hvor dypt vi skulle traversere ned i quadtreet for hver bilderamme må vi ha et avstandsmål ϵ_t. Denne var gitt som vinkelen til en piksel ganger avstanden fra midpunktet i hver tile til kameraet. Siden kameraet kan forflytte seg for hver bilderamme måtte vi beregne avstandsmålet ϵ_t for hver traversering i quadtreet.

Den rekursive traverseringsprosessen undersøkte for hver tile t om:

$$\epsilon_t < \delta_t. \tag{6.2}$$

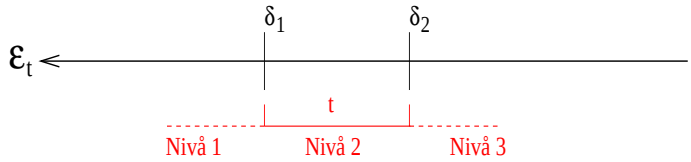
Hvis ligning (6.2) var tilfredsstilt, med andre ord hvis avstanden ϵ_t til tilen t var mindre enn objektfeilen δ_t, skulle vi traversere videre ned i barna til t.

Vi skal nå se nærmere på relasjonen mellom anstandsmålet ϵ_t og objektfeilen δ_t. Dette er viktig fordi vi utnytter denne under visualisering. Fra ligning (6.2) kan vi slå fast at vi ikke skal traversere videre ned i barna til t så lenge:

$$\epsilon_t \geq \delta_t. \tag{6.3}$$

For en tile t kan vi dermed slå fast at med en gang avstanden ϵ_p til en foreldretile p er slik at $\epsilon_p < \delta_p$, skal vi traversere til t.

Vi kan illustrere dette ved figur 6.1. Tenk at vi har gitt avstandsmålet ϵ_t til en tile t på nivå 2 i quadtreet som på figuren. Vi traverserer til t med en gang avstandsmålet $\epsilon_p < \delta_1$ for foreldretilen p. Vi skal ikke traversere videre ned i t så lenge $\epsilon_t \geq \delta_2$.



Figur 6.1: *Eksempel på traverseringen til en tile t på nivå 2 i quadtreet.*

For å forenkle visualiseringsprosessen sier vi at traverserte til t når $\epsilon_t < \delta_1$ på figur 6.1. Generelt kan vi dermed si at vi skal traversere i t så lenge:

$$\delta_t \leq \epsilon_t < \delta_p, \tag{6.4}$$

altså så lenge avstanden ϵ_t er mindre enn objektfeilen til foreldretilen p og større enn objektfeilen til t. Vi skal komme tilbake til dette i seksjon 6.3.

Vi har to særtilfeller som vi må ta hensyn til:

- *Tilen t er rotnoden ($l_t = 0$)*: Siden rotnoden ikke har noen foreldre kan vi aldri traversere videre opp i denne.
- *Tilen t ligger på det maksimale dypeste nivået l_{maks} i quadtreet*: Siden tiler på nivå l_{maks} ikke har noen barn kan vi heller ikke traversere dypere ned i dem.

I seksjon 5.2.2 introduserte vi våre tre $LOD_i, i = 0, 1, 2$ grenser. Disse ble forbundet med de tre nivåene i vårt diskret LOD hierarki for trerepresentasjonene: LOD_0 bestemmer når vi skal tegne nivå 0 (planrepresentasjonene),

LOD_1 bestemmer når vi skal tegne nivå 1 (kryss-planrepresentasjonene) og LOD_2 bestemmer når vi skal tegne nivå 2 (stjerne-planrepresentasjonene). Vi er garantert at:

$$LOD_0 > LOD_1 > LOD_2 \tag{6.5}$$

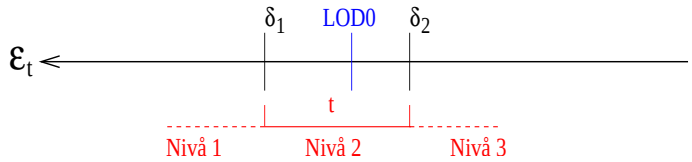
De ulike LOD_i verdiene ble knyttet mot avstandsmålet ϵ_t til en tile t ved å velge den $i'te$ trerepresentasjonen dersom:

$$LOD_{i+1} < \epsilon_t \leq LOD_i \tag{6.6}$$

Også her har vi to særtilfeller som vi må ta hensyn til:

- $\epsilon_t > LOD_0$: Om avstandsmålet ϵ_t er større enn LOD_0 skal vi ikke tegne noen trær i t. Dette tilsvarer at vi ikke ønsker å tegne trær før vi er nærme nok terrenget.
- $\epsilon_t \leq LOD_2$: Siden stjerne-planrepresentasjonen er den meste detaljerte trerepresentasjonen i vår modell, kan vi ikke velge noen finere trerepresentasjon enn denne.

Figur 6.2 viser et eksempel på hvordan vi kan illustrere sammenhengen mellom quadtretraverseringen og de ulike LOD_i grensene. Har har vi gitt samme tile t på nivå 2 som på figur 6.1, men vi har nå i tillegg lagt til LOD_0 grensen slik at $\delta_2 < LOD_0 < \delta_1$. Dette vil si at vi under opptegningen av t skal tegne planrepresentasjonene for alle utplasserte trær i t.



Figur 6.2: *Eksempel på hvordan vi knytter sammen LOD_0 grensen med traverseringen til en tile t på nivå 2 i quadtreet. Så lenge $\epsilon_t > LOD_0$ skal vi ikke tegne noen trær. Derimot skal vi tegne planrepresentasjonene for alle trær i t når $\epsilon_t \leq LOD_0$.*

For å forhindre at trær plutselig popper fram i landskapet med en gang $\epsilon_t \leq LOD_0$ som på figur 6.2 bruker vi alfablending. Vi bruker også alfablending for å glatte ut skiftene mellom våre trerepresentasjoner, slik at de nye plan som innføres for hver trerepresentasjon ikke plutselig dukker opp. For å gjøre dette husker vi fra seksjon 5.2.3 at vi måtte ha en blendinglengde *trans*. Blendinglengden *trans* bestemmer over hvor lang avstand vi skal blende inn trærne i en tile t. Denne avstanden er også gitt av avstandsmålet ϵ_t.

For å sørge for glatt blending, innførte vi en blendingfunksjon $f_i(\epsilon_t)$ gitt som:

$$f_i(\epsilon_t) = \frac{LOD_i - \epsilon_t}{trans} \tag{6.7}$$

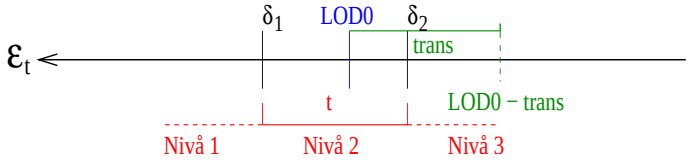
Denne sørger for å blende inn den $i'te$ trerepresentasjonen fra $\epsilon_t = LOD_i$ til $\epsilon_t = LOD_i - trans$. Vi minner om at en alfaverdi på 0 betyr at objektet med denne alfaverdien vil være gjennomsiktig. En alfaverdi på 1 betyr derimot at objektet er opak. Alfaverdier mellom 0 og 1 angir ulike grad av gjennomskinnelighet.

Vi innførte videre α_{LOD_0} som alfaverdien til planrepresentasjonene i en tile t, α_{LOD_1} som alfaverdien til kryss-planrepresentasjonene og α_{LOD_2} som alfaverdien til stjerne-planrepresentasjonene. Vi satt:

$$\alpha_{LOD_0} = f_0(\epsilon_t) \tag{6.8}$$
$$\alpha_{LOD_1} = f_1(\epsilon_t) \tag{6.9}$$
$$\alpha_{LOD_2} = f_2(\epsilon_t), \tag{6.10}$$

for å sørge for riktig blending av de ulike trepresentasjonene.

Figur 6.3 viser blendingsprosessen for samme tile t som på figur 6.2. Blendingen av planrepresentasjonene i t skal altså skje fra $\epsilon_t = LOD_0$ til $\epsilon_t = LOD_0 - trans$. Ved å velge $\alpha_{LOD_0} = f_0(\epsilon_t)$, vil denne sørge for at $\alpha_{LOD_0} = 0$ når $\epsilon_t = LOD_0$ og $\alpha_{LOD_0} = 1$ når $\epsilon_t = LOD_0 - trans$. Etterhvert som avstanden til t minker fra $\epsilon_t = LOD_0$ økes α_{LOD_0} tilsvarende. Dette fører til at alle planrepresentasjonene i t blir jevnt blendet inn i terrenget.



Figur 6.3: *Eksempel på hvordan knytter blendingen av en trerepresentasjon til en tilen på figur 6.2. Planrepresentasjonen skal blendes inn fra* $\epsilon_t = LOD_0$ *til* $\epsilon_t = LOD_0 - trans$.

Figur 6.3 illustrerer også et problem som kan forekomme under visualiseringen. Her ser vi at $\delta_2 > LOD_0 - trans$ som vil føre til at vi planrepresentasjonene i t ikke blir ferdig blendet inn før vi må traversere ned i barna til t. Vi skal ta for oss denne problemstillingen når vi introduserer våre to visualiseringsmetoder.

Vi er garantert at:

$$LOD_0 - trans > LOD_1, \tag{6.11}$$

og

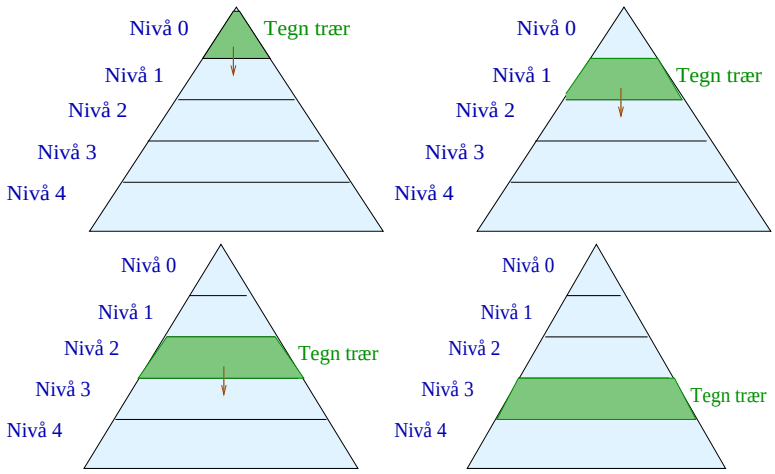
$$LOD_1 - trans > LOD_2. \tag{6.12}$$

Dette fører til at vi alltid kan forsikre oss om at *trans* aldri blir for stor til at forrige trepresentasjon ikke er ferdig blendet inn i landskapet før vi skal begynne med å blende inn en ny trerepresentasjon.

6.2 Visualisering av skogmodellen

Vi er nå klare for å se på hvordan vi skal visualisere skogmodellen vår. Vi ser på to ulike metoder: direkterendering og dybderendering. Begge metodene er knyttet til hvordan vi traverserer quadtreet.

For direkterenderingsmetoden tegner vi trærne for hver tile t etterhvert som vi traverserer dypere ned i quadtreet. For å hjelpe oss å illustrere dette kan det være hensiktsmessig å tenke på quadtreet som en pyramide. Anta at vi har gitt et avstandsmål ϵ_t slik at vi for en bilderamme skal traversere ned til en tile t på nivå 3 som vist på figur 6.4. Under traverseringen ned til nivå 3, tegner vi først trærne i rotnoden. Så traverserer vi ned til barna til rotnoden på nivå 1 og tegner trærne i disse. Vi traverserer videre ned i til nivå 2 og tegner trærne i disse. Til slutt traverserer vi ned til tilen på nivå 3 og tegner trærne i denne. Siden vi ikke skal traversere videre ned i quadtreet, er dermed traverseringen for denne bilderamme ferdig.



Figur 6.4: *Et eksempel på direkterendering. Vi er gitt ϵ_t slik at vi må traversere ned til en tile på nivå 3 på figuren. Dette fører til fire separate steg under traverseringsprosessen.*

Direkterendering er dermed gitt ved følgende algoritme:

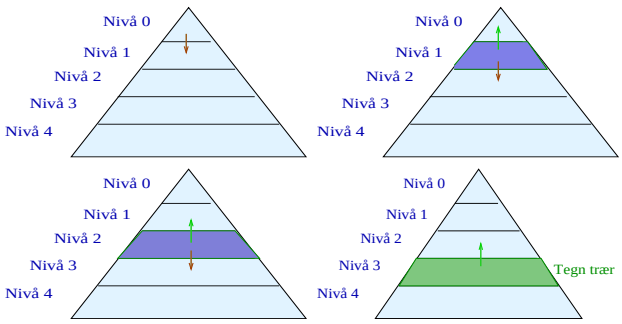
Algoritme 6.1 traverserMedDirekteRendering(t)

```
1  hvis B_t ikke befinner seg innenfor synsvolumet
2      avslutt traverseringen for t
3  τ_t = curr_frame
4  hvis μ_t ≠ 0
5      tegnMedDirekterendering( t )
6      hvis δ_t > ε_t og l_t < l_maks
7          for alle barn b til t
8              hvis b ikke er instansiert
9                  instansier b
10             traversere( b )
```

Eneste forskjell mellom algoritme 6.1 og algoritme 4.14 på side 53 er kallet på *tegnMedDirekterendering(t)* på linje 5. Hvis antallet trær μ_t utplassert i en tile t er forskjellig fra 0 skal vi tegne trærne i t før vi eventuelt traverserer videre ned i barna til t.

For dybderendering tegner vi ikke trær i en tile før vi under traverseringsalgoritmen finner ut at vi har nådd en tile t der vi ikke skal traversere videre ned i barna til t. Altså, for et gitt avstandsmål ϵ_t, traverserer vi quadtreet til vi finner en tile t der $\epsilon_t \geq \delta_t$. Siden vi ikke skal traversere dypere ned i t, tegner vi trærne i denne tilen.

For å forsikre oss om at vi samtidig tegner alle utplasserte trær i tilene over t i quadtreet, må vi duplisere treposisjonene fra foreldretilen som forklart i seksjon 4.2.3. Hver tile t inneholder dermed både sine egne utplasserte treposisjoner samt treposisjonene i foreldretilen som ligger innenfor grensene til t. Tegner vi trærne i en tile t tegner vi dermed alle disse.



Figur 6.5: *Et eksempel på dybderendering. Vi er gitt ϵ_t slik at vi må traversere ned til en tile på nivå 3 på figuren. Vi skal dermed ikke tegne trær før vi når t. Blått betyr at tilen på det tilhørende nivået har duplisert treposisjonene fra nivået over i quadtreet.*

Anta at vi skal traversere ned til nivå 3 på figur 6.5. For rotnoden gjør vi ikke annet enn å traversere videre ned til barna på nivå 1. For de fire barna til rotnoden på nivå 1 har hver av dem duplisert treposisjonene til rotnoden som ligger innenfor sine grenser og utplassert sine egne trær etter at de ble instansiert. Tilsvarende for tiler på nivå 2 og 3. Siden vi ikke skal traversere videre ned i quadtreet skal vi tegne vi trærne i t. Dette fører til at t tegner både sine egne trær og alle trær som ble utplassert i de tre nivåene over t (som ligger innenfor t).

Traverseringsalgoritmen for dybderendering blir dermed forskjellig enn for direkterendering. I stedet for å tegne trær i alle tiler før vi traverserer videre, skal vi vente med å tegne trær helt til vi ikke skal traversere dypere ned i quadtreet. Dybderendering er gitt ved følgende algoritme:

Algoritme 6.2, traverserMedDybdeRendering(t)

```
1  hvis B_t ikke befinner seg innenfor synsvolumet
2      avslutt traverseringen for t
3  τ_t = curr_frame
4  hvis μ_t ≠ 0
5      tegnMedDirekterendering( t )
6      hvis δ_t > ε_t og l_t < l_maks
7          for alle barn b til t
8              hvis b ikke er instansiert
9                  instansier b
10             traversere( b )
11     ellers
12         tegnMedDybderendering( t )
```

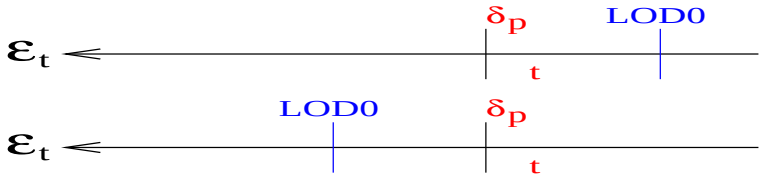
Vi er nå klare for å beskrive de ulike visualiseringsmetodene i detalj.

6.2.1 Direkterendering

For å beskrive opptegningen av trær for direkterenderingsmetoden ser vi kun på en tile t. Dette gjøres fordi tilsvarende prosess gjøres for alle tiler i quadtreet. Vi kan hovedsaklig dele direkterenderingsmetoden i to deler: tiler hvor $\delta_p \geq LOD_0$ og tiler hvor $\delta_p < LOD_0$.

Figur 6.6 viser de to tilfellene. Vi husker at vi traverste til en tile t når $\epsilon_t < \delta_p$; altså når avstandsmålet ϵ_t er mindre enn objektfeilen δ_p til foreldretilen p. Grunnen til at vi må skille mellom de to tilfellene er at for alle tiler der $\delta_p < LOD_0$, som i det nederste tilfellet på figur 6.6, har allerede LOD_0 skiftet forekommet. Dette vil si at trær har allerede blitt blendet inn i foreldretilen fordi vi ønsker å blende inn trær fra $\epsilon_t = LOD_0$.

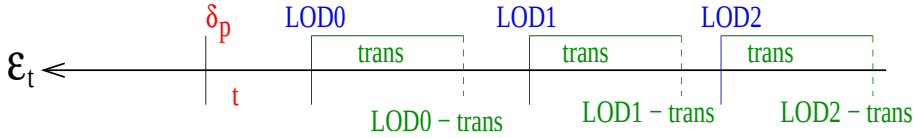
Vi konsentrerer oss om tilfellet hvor $\delta_p \geq LOD_0$ først. Her har ikke LOD_0 skiftet forekommet enda; vi skal altså blende inn trær etter at vi har



Figur 6.6: *Vi har hovedsaklig to separate tilfeller som kan skje for en tile under direkterendering: $\delta_p \geq LOD_0$ (øverste tilfelle) eller $\delta_p < LOD_0$ (nederste tilfelle).*

traversert til t. Husk at vi for direkterenderingsmetoden tegner alle tiler hver for seg: en tile t tegner sine trær før vi eventuelt skal traversere videre ned i barna til t. Dette vil si at hver tile t blender inn sine egne trær uavhengig de andre tilene i quadtreet.

Gitt en blendinglengde *trans* kan vi da blende inn trærne i en tile t på vanlig måte. Hvis $\epsilon_t > LOD_0$ skal vi ikke tegne noen trær. Med en gang avstandsmålet ϵ_t for blir så liten at $\epsilon_t = LOD_0$ begynner vi blendingen med å sette $\alpha_{LOD_0} = f_0(\epsilon_t)$. Denne vil da sørge for at at planrepresentasjonene i t blir jevnt blendet inn til $\epsilon_t = LOD_0 - trans$. På samme måte blender vi inn kryss-planrepresentasjonene fra $\epsilon_t = LOD_1$ til $\epsilon_t = LOD_1 - trans$ og stjerne-planrepresentasjonene fra $\epsilon_t = LOD_2$ til $\epsilon_t = LOD_2 - trans$.



Figur 6.7: *For en tile t hvor $\delta_p \geq LOD_0$ kan vi blende inn trær på normal måte. Vi blender inn den i'te trerepresentasjonen fra $\epsilon_t = LOD_i$ til $\epsilon_t = LOD_i - trans$. Om $\epsilon_t > LOD_0$ skal vi ikke tegne trær.*

Figur 6.7 viser dette. Siden LOD_0 skiftet ikke har forekommet før vi traverserer ned i t kan vi blende inn trær på vanlig måte. Før vi gir algoritmen for å tegne trær med direkterenderingsmetoden minner vi om at hver tile t har samlet alle trerepresentasjonene sine i tre ulike "display lists". Alle planrepresentasjonene for t ligger i $list_0^t$, kryss-planrepresentasjonene i $list_1^t$ og stjerne-planrepresentasjonene ligger i $list_2^t$. Å tegne trærne i en tile t består dermed i å eksekvere de ulike listene med en gitt alfaverdi. Dette vil sørge for at alle trerepresentasjonene i t blir tegnet med denne alfaverdien.

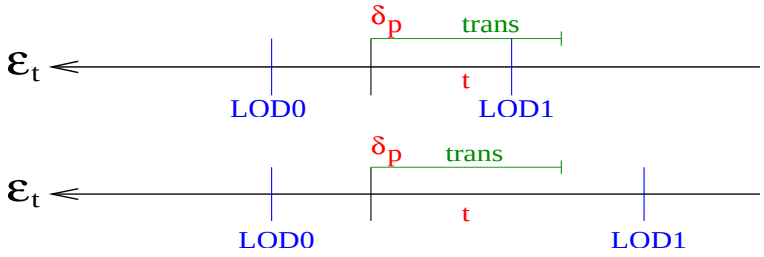
Vi er nå klare for å gi første halvdel av *tegnMedDirekterendering(t)* metoden. Vi ignorerer altså foreløpig tilfellet hvor $\delta_p < LOD_0$ for en tile t. Vi gir denne ved følgende algoritme:

Algoritme 6.3, tegnMedDirekterendering(t) [I]

```
1  hvis ε_t > LOD_0
2      avslutt tegnMedDirekterendering( t )
3  hvis δ_p ≥ LOD_0
4      hvis LOD_0 ≥ ε_t > LOD_1
5          eksekver list_0^t med α_LOD0 = f_0(ε_t)
6      ellers hvis LOD_1 ≥ ε_t > LOD_2
7          eksekver list_0^t med α_LOD0 = 1.0
8          eksekver list_1^t med α_LOD1 = f_1(ε_t)
9      ellers hvis ε_t ≤ LOD_2
10         eksekver list_0^t med α_LOD0 = 1.0
11         eksekver list_1^t med α_LOD1 = 1.0
12         eksekver list_2^t med α_LOD2 = f_2(ε_t)
```

Siden vi er garantert at forrige trerepresentasjon er ferdig blendet inn før vi skal velge en ny trerepresentasjon, kan vi da tegne forrige trerepresentasjon med en alfaverdi 1. For eksempel for kryss-planrepresentasjonen vet vi at $\alpha_{LOD_0} \geq 1$ når $\epsilon_t = LOD_1$ så vi kan dermed tegne alle planrepresentasjonene med $\alpha_{LOD_0} = 1$. Vi ser også at dersom $\epsilon_t > LOD_0$ skal vi ikke tegne trær i t. Vi avslutter dermed *tegnMedDirekteRendering(...)*.

Vi er nå klare for å behandle de ulike tilfellene for en tile t der $\delta_p < LOD_0$. Figur 6.8 viser de to tilfellene som kan forekomme for en tile t der $\delta_p < LOD_0$. Vi konsentrerer oss om å blende inn planrepresentasjonene i t, da tilsvarende prosess brukes for å blende inn kryss- og stjerneplanrepresentasjonene.



Figur 6.8: *Tilfellene for en tile t der $\delta_p < LOD_0$.*

For å blende inn planrepresentasjonene våre ønsker vi at $\alpha_{LOD_0} = 0$ når $\epsilon_t = \delta_p$ for en tile t der $\delta_p < LOD_0$. Vi husker at at vi traverserte til t når $\epsilon_t < \delta_p$. Hvis vi forsøker å blende inn planrepresentasjonene med $f_0(\epsilon_t)$ som i ligning (6.7) for t, vil vi oppleve at blendingen blir feil fordi $f_0(\epsilon_t) = 0$ når $\epsilon_t = LOD_0$. Siden $\delta_p < LOD_0$, vil dermed $f_0(\delta_p) > 0$ for en tile t. Dermed vil ikke planrepresentasjonene bli glatt blendet inn for tilen t og vi vil oppleve uønskede visuelle effekter.

Vi må altså innføre en ny blendingsfunksjon $g_{\delta_p}(\epsilon_t)$ som er slik at:

$$g_{\delta_p}(\epsilon_t) = \begin{cases} 0, & \epsilon_t = \delta_p \\ 1, & \epsilon_t = \delta_p - trans \end{cases} \tag{6.13}$$

Vi ønsker altså fortsatt å blende over samme blendinglengde, men blendingfunksjonen må være 0 fra vi traverserer ned i en tile t ($\epsilon_t < \delta_p$). Vi kan dermed gi $g_{\delta_p}(\epsilon_t)$ som:

$$g_{\delta_p}(\epsilon_t) = \frac{\delta_p - \epsilon_t}{trans} \tag{6.14}$$

Vi ser av ligning (6.14) at:

$$g_{\delta_p}(\delta_p) = \frac{\delta_p - \delta_p}{trans} = 0 \tag{6.15}$$

og

$$g_{\delta_p}(\delta_p - trans) = \frac{\delta_p - (\delta_p - trans)}{trans} = 1 \tag{6.16}$$

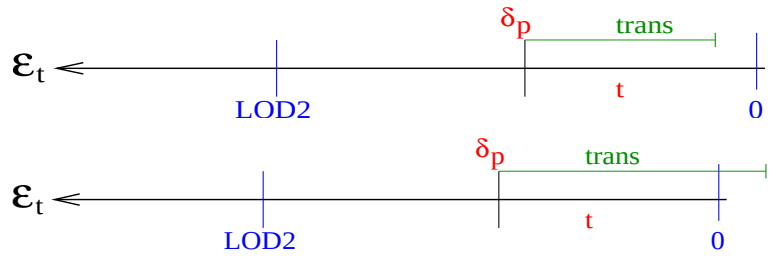
Som før vil synkende verdier for ϵ_t fra δ_p til $\delta_p - trans$ gi en tilsvarende økning av alfaverdien fra 0 til 1.

Ved å blende inn planrepresentasjonene fra $\epsilon_t = \delta_p$ til $\epsilon_t = \delta_p - trans$ vil vi få tilfellet hvor kryss-planrepresentasjonen, som velges av vår LOD seleksjonsmekanismen når $\epsilon_t = LOD_1$, blendes oppå planrepresentasjonen kanskje ikke har rukket å oppnå $\alpha_{LOD_0} = 1$ enda. Dette vil forekomme dersom $\delta_p - trans < LOD_1$ som vist øverst på figur 6.8. Uansett blendes kryss-planrepresentasjonen med $\alpha_{LOD_1} = f_1(\epsilon_t)$, så $\alpha_{LOD_1} = 0$ når $\epsilon_t = LOD_1$. Det spiller dermed ingen rolle om alfaverdien til planrepresentasjonen ikke har rukket å bli 1, siden $\alpha_{LOD_0} = g_{\delta_p}(\epsilon_t)$ vil uansett bli 1 før $\alpha_{LOD_1} = f_1(\epsilon_t)$ siden vi fortsatt blender over den samme blendingslengden *trans*.

På nøyaktig samme måte som $\delta_p < LOD_0$ kan forekomme for en tile t, kan vi oppleve at $\delta_p < LOD_1$. Dette er identisk med situasjonen på figur 6.8 bortsett fra vi bytter LOD_0 med LOD_1 og LOD_1 med LOD_2 på figuren. Vi nå da må blende inn både plan- og kryss-planrepresentasjonen med $g_{\delta_p}(\epsilon_t)$ fordi vi ingen av disse har blitt tegnet før i t.

Tilfellet hvor $\delta_p < LOD_2$ for en tile t er spesielt. De to tilfellene som kan forekomme er vist på figur 6.9. Vi vet at avstandsmålet ϵ_t aldri kan bli mindre enn 0 til en tile t, men hva skjer for en tile dersom $\delta_p - trans < 0$ som nederst på figur 6.9? Vi vil da få trær som aldri blendes fullstendig inn i landskapet om vi blender inn med $g_{\delta_p}(\epsilon_t)$ og vi vil oppleve halvgjennomsiktige trær under visualisering.

For å forhindre dette, må vi for hver tile t der $\delta_p < LOD_2$, undersøke om $\delta_p - trans < 0$. Hvis $\delta_p - trans \geq 0$ kan vi blende inn med $g_{\delta_p}(\epsilon_t)$ som i ligning (6.14). Ellers søker vi en ny blendingfunksjon $h_{\delta_p}(\epsilon_t)$ som er slik at:



Figur 6.9: *Tilfellet for en tile t der* $\delta_p < LOD_2$. *Hvis* $\delta_p - trans \geq 0$ *kan vi sette* $\alpha_{LOD_0} = \alpha_{LOD_1} = \alpha_{LOD_2} = g_{\delta_p}(\epsilon_t)$. *Ellers må vi sette* $\alpha_{LOD_0} = \alpha_{LOD_1} = \alpha_{LOD_2} = h_{\delta_p}(\epsilon_t)$ *som i ligning (6.18).*

$$h_{\delta_p}(\epsilon_t) = \begin{cases} 0 & , \epsilon_t = \delta_p \\ 1 & , \epsilon_t = 0. \end{cases} \tag{6.17}$$

Vi kan dermed gi $h_{\delta_p}(\epsilon_t)$ som:

$$h_{\delta_p}(\epsilon_t) = \frac{\delta_p - \epsilon_t}{\delta_p}. \tag{6.18}$$

Vi ser at $h_{\delta_p}(\epsilon_t)$ vil gi glatt blending siden:

$$h_{\delta_p}(\delta_p) = \frac{\delta_p - \delta_p}{\delta_p} = 0, \tag{6.19}$$

og

$$h_{\delta_p}(0) = \frac{\delta_p - 0}{\delta_p} = 1. \tag{6.20}$$

Ved å blende inn med $h_{\delta_p}(\epsilon_t)$ for tiler der $\delta_p < LOD_2$ og $\delta_p - trans < 0$ vil vi dermed få glatt blending fra $\epsilon_t = \delta_p$ til $\epsilon_t = 0$.

Vi er dermed forsikret om at alle trær i terrenget blir glatt blendet inn dersom $\delta_p < LOD_0$ for en tile t i quadtreet. Vi er nå klare for å gi andre halvdel av *tegnMedDirekterendering(t)* for alle tiler t der $\delta_p < LOD_0$. Vi gir denne som følger:

Algoritme 6.4, tegnMedDirekterendering(t) [II]

```
13 ellers hvis δ_p < LOD_0
14     hvis δ_p < LOD_0 og δ_p ≥ LOD_1
15         hvis LOD_0 ≥ ε_t > LOD_1
16             eksekver list_0^t med α_{LOD_0 = g_{δ_p}(ε_t)}
17         ellers hvis LOD_1 ≥ ε_t > LOD_2
18             eksekver list_0^t med α_{LOD_0} = g_{δ_p}(ε_t)
19             eksekver list_1^t med α_{LOD_1} = f_1(ε_t)
20         ellers hvis ε_t ≤ LOD_2
21             eksekver list_0^t med α_{LOD_0} = 1.0
22             eksekver list_1^t med α_{LOD_1} = 1.0
23             eksekver list_2^t med α_{LOD_2} = f_2(ε_t)
24     ellers hvis δ_p < LOD_1 og δ_p ≥ LOD_2
25         hvis ( LOD_1 ≥ ε_t > LOD_2 )
26             eksekver list_0^t med α_{LOD_0} = g_{δ_p}(ε_t)
27             eksekver list_1^t med α_{LOD_1} = g_{δ_p}(ε_t)
28         ellers hvis ε_t ≤ LOD_2
29             eksekver list_0^t med α_{LOD_0} = g_{δ_p}(ε_t)
30             eksekver list_1^t med α_{LOD_1} = g_{δ_p}(ε_t)
31             eksekver list_2^t med α_{LOD_2} = f_2(ε_t)
32     ellers hvis δ_p < LOD_2)
33         hvis ( δ_p − trans ≥ 0 )
34             eksekver list_0^t med α_{LOD_0} = g_{δ_p}(ε_t)
35             eksekver list_1^t med α_{LOD_1} = g_{δ_p}(ε_t)
36             eksekver list_2^t med α_{LOD_2} = g_{δ_p}(ε_t)
37         ellers
38             eksekver list_0^t med α_{LOD_0} = h_{δ_p}(ε_t)
39             eksekver list_1^t med α_{LOD_1} = h_{δ_p}(ε_t)
40             eksekver list_2^t med α_{LOD_2} = h_{δ_p}(ε_t)
```

For å beskrive denne algoritmen holder det at vi tar for oss ett eksempel. Vi velger å se på tilfellet hvor $\delta_p < LOD_0$ og $\delta_p \geq LOD_1$ som på linje 15 i algoritmen over. Vi har da tilfellet på figur 6.8; vi skal blende planrepresentasjonene i t fra $\epsilon_t = \delta_p$. Vi husker at $g_{\delta_p}(\epsilon_t)$ som i ligning (6.14) tilfredsstiller dette.

Så lenge $LOD_0 \geq \epsilon_t > LOD_1$ er ϵ_t innenfor grensene for å tegne planrepresentasjonene i en tile t. Vi velger da $\alpha_{LOD_0} = g_{\delta_p}(\epsilon_t)$ og eksekverer $list_0^t$ med denne alfaverdien. Dette vil sørge for å jevnt blende planrepresentasjonene inn i terrenget fra vi traverserer ned i t.

Siden vi ikke er garantert at blendingen er ferdig når $\epsilon_t = LOD_1$, som i det øverste tilfellet på figur 6.8, må vi fortsette å blende planrepresentasjonene med $g_{\delta_p}(\epsilon_t)$ som på linje 18 i algoritmen 6.4. Siden skiftet for kryss-planrepresentasjonene ikke har forekommet for t siden at $\delta_p \geq LOD_1$,

kan vi blende inn kryss-planrepresentasjonen med $\alpha_{LOD_1} = f_1(\epsilon_t)$. Vi kan dermed eksekverer $list_1^t$ med denne alfaverdien.

For å blende inn stjerne-planrepresentasjonen, velger vi $\alpha_{LOD_2} = f_2(\epsilon_t)$. Siden vi er garantert at $LOD_1 - trans > LOD_2$ er kryss-planrepresentasjonen ferdig blendet inn når $\epsilon_t = LOD_2$. Siden vi blender inn for samme blendinglengde, er også planrepresentasjonene ferdig blendet inn i tilen. Dermed kan vi eksekvere både $list_0^t$ og $list_1^t$ med alfaverdi 1.0.

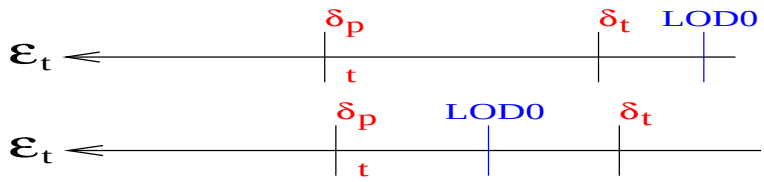
6.2.2 Dybderendering

For dybderendering tegner vi kun trærne i den dypeste tilen vi traverserer til i quadtreet. Det første vi kan merke oss da er at vi for tiler der $\delta_t > LOD_0$ ikke behøver å initialisere noen "display lists". Vi husker at vi traverserer i en tile t så lenge:

$$\delta_t \leq \epsilon_t < \delta_p. \tag{6.21}$$

Med en gang $\epsilon_t < \delta_t$ skal vi traversere videre ned i barna til t.

Dersom vi for en tile t har at $\delta_t > LOD_0$, vil ikke LOD_0 skiftet forekomme før vi må traversere videre ned i barna til t. Om vi skal traversere ned i barna til t, skal vi ifølge dybderenderingsmetoden ikke lenger tegne t. Figur 6.10 viser de to tilfellene. Vi kan dermed se bort ifra alle tiler der $\delta_t > LOD_0$.



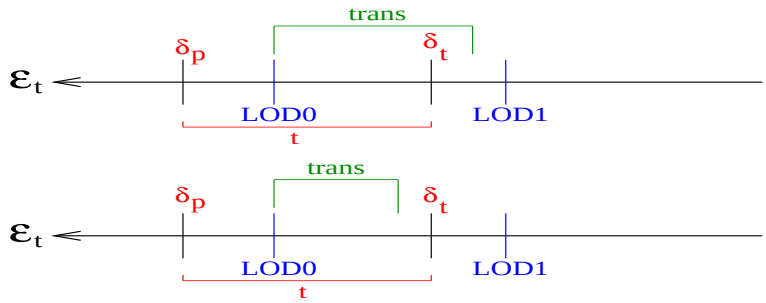
Figur 6.10: *Det øverste tilfellet viser en tile der $\delta_t > LOD_0$. Siden vi aldri skal tegne trær i t før vi skal traversere ned i barna til t, er det heller ingen grunn til å generere "display lists" for denne tilen for dybderenderingsmetoden. Derimot, om $\delta_t \leq LOD_0$ som i det nederste tilfellet, skal vi tegne trær i t.*

Blendingen av trær i quadtreet for dybderenderingsmetoden er mer komplisert enn for direkterendering. Dette er fordi alle nivåer under i quadtreet er avhengig av nivået over. Vi kan midlertidig forenkle visualiseringen noe ved å kreve at alle trærne i en tile t skal ha alfaverdi 1 når $\epsilon_t = \delta_t$. Skal vi traversere videre ned i barna b til t, kan vi dermed se bort i fra blendingen av de dupliserte treposisjonene fra t i hver b. De dupliserte treposisjonene kan i stedet tegnes med alfaverdi 1 siden vi er garantert at disse er 1 før vi traverserer ned i b.

Vi må igjen skille mellom det samme tilfellet som for direkterenderingsmetoden: $\delta_p \geq LOD_0$ og $\delta_p < LOD_0$. Med andre ord; tilfellet der vi for

første gang blender inn trær i en tile t og tilfellet hvor trær allerede har blitt blendet inn i foreldretilen. Vi konsentrerer oss om tilfellet hvor $\delta_p \geq LOD_0$ først. Dette vil tiler der LOD_0 skiftet ikke har forekommet i foreldretilen til t.

Figur 6.11 viser de to tilfellene der LOD_0 skiftet ligger mellom δ_p og δ_t for en tile og $LOD_1 < \delta_t$. Det er verdien til blendinglengden *trans* som bestemmer hvem av de to tilfellene som vil forekomme. Er $LOD_0 - trans \geq \delta_t$ som nederst på figur 6.11 kan vi blende inn trerepresentasjonene med $f_0(\epsilon_t)$ som i ligning (6.7). Dette kan gjøres fordi $f_0(\epsilon_t)$ vil sørge for at alfaverdiene til planrepresentasjonene i t er 1 når $\epsilon_t = \delta_t$. Om $LOD_0 - trans < \delta_t$, må vi blende inn med en ny blendingfunksjon.



Figur 6.11: *De to tilfellene som kan forekomme for en tile t der LOD_0 skiftet ligger mellom δ_p og δ_t. I dette tilfellet er $LOD_1 < \delta_t$.*

Blendinglengden for det øverste tilfellet er for stor. Siden brukeren kan selv velge LOD_i og *trans* verdi, har vi ingen garanti for at tilfellet til øverst ikke kan forekomme. Fordi vi krever at alfaverdien til alle trærne i t skal være 1 når $\epsilon_t = \delta_t$ må vi innfør en ny blendingfunksjon.

Siden vi ikke kan lenger blende over vår gitte blendinglengde, innfører vi en generell blendingfunksjon $m_{[l_1,l_2]}(\epsilon_t)$ gitt ved:

$$m_{[l_1,l_2]}(\epsilon_t) = \begin{cases} 0, & \epsilon_t = l_1 \\ 1, & \epsilon_t = l_2 \end{cases} \tag{6.22}$$

Vi kan dermed gi $m_{[l_1,l_2]}(\epsilon_t)$ på formen:

$$m_{[l_1,l_2]}(\epsilon_t) = \frac{l_1 - \epsilon_t}{l_1 - l_2}. \tag{6.23}$$

Denne blendingfunksjonen vil sørge for å blende inn trær fra $\epsilon_t = l_1$ til $\epsilon_t = l_2$. Den nye blendinglengden er dermed gitt ved $l_1 - l_2$. Vi bruker denne blendingfunksjonen ved å gi ulike parametre for l_1 og l_2.

For tilfellet øverst på figur 6.11 der $LOD_0 - trans < \delta_t$ må vi velge $\alpha_{LOD_0} = m_{[LOD_0,\delta_t]}(\epsilon_t)$. Dette vil gi:

$$m_{[LOD_0,\delta_t]}(\epsilon_t) = \frac{LOD_0 - \epsilon_t}{LOD_0 - \delta_t}. \tag{6.24}$$

som gir at:

$$m_{[LOD_0,\delta_t]}(LOD_0) = \frac{LOD_0 - LOD_0}{LOD_0 - \delta_t} = 0. \tag{6.25}$$

og

$$m_{[LOD_0,\delta_t]}(\delta_p) = \frac{LOD_0 - \delta_t}{LOD_0 - \delta_t} = 1. \tag{6.26}$$

som garanterer at planrepresentasjonene i t får en alfaverdi på 1 før vi traverserer ned i barna til t.

Vi har ingen garanti for at $LOD_1 < \delta_t$ som på figur 6.11. Siden det er brukeren som selv bestemmer verdiene til LOD_i i skogkonfigurasjonsfilen kan også $\delta_p \leq LOD_1 \leq \delta_t$. Det eneste vi har garanti for er at $LOD_0 > LOD_1$ og at $LOD_0 - trans > LOD_1$ som nevnt i seksjon 5.2.3. Tilsvarende gjelder også for LOD_2.

Siden alle tiler der $\delta_p \geq LOD_0$ ikke har tegnet noen trær i foreldretilen, må vi blende inn både trærne som tilhører t og de dupliserte treposisjonene i t. For å ta hensyn til dette må nå hver tile t nå inneholde seks "display lists". Som før bruker vi $list_i^t$ som betegnelsen på listen for alle trærne som ble utplassert i t. For dybderendering må vi i tillegg introdusere $list_i^p$ for de ulike trerepresentasjonene i t som ble duplisert fra foreldretilen p. Hver av de seks ulike listene må ha sin egen alfaverdi. Vi innfører $\alpha_{LOD_i}^t$ for alfaverdiene til $list_i^t$ og $\alpha_{LOD_i}^p$ for alfaverdiene til $list_i^p$.

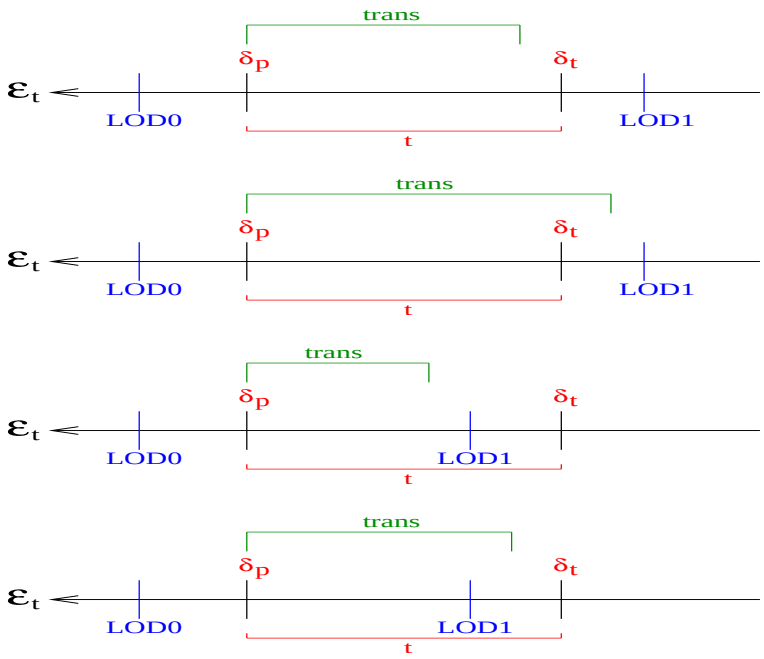
For tiler der $\delta_p \geq LOD_0$ er altså $\alpha_{LOD_i}^t = \alpha_{LOD_i}^p$. Dette vil si at alle trærne i t skal blendes for samme alfaverdi. Vi er dermed klare for å gi første halvdel av *tegnMedDybderendering(t)*:

Algoritme 6.5, tegnMedDybderendering(t) [I]

```
1  hvis ε_t > LOD_0
2     avslutt tegnMedDybderendering( t )
3  hvis δ_p ≥ LOD_0
4     hvis LOD_0 ≥ ε_t > LOD_1
5        hvis LOD_0 − trans ≥ δ_t
6           eksekver list_0^t med α_{LOD_0}^t = f_0(ε_t)
7           eksekver list_0^p med α_{LOD_0}^p = α_{LOD_0}^t
8        ellers
9           eksekver list_0^t med α_{LOD_0}^t = m_{[LOD_0,δ_t]}(ε_t)
10          eksekver list_0^p med α_{LOD_0}^p = α_{LOD_0}^t
11    ellers hvis LOD_1 ≥ ε_t > LOD_2
12       eksekver list_0^t med α_{LOD_0}^t = 1.0
13       eksekver list_0^p med α_{LOD_0}^p = 1.0
14       hvis LOD_1 − trans ≥ δ_t
15          eksekver list_1^t med α_{LOD_1}^t = f_1(ε_t)
16          eksekver list_1^p med α_{LOD_1}^p = α_{LOD_1}^t
17       ellers
18          eksekver list_1^t med α_{LOD_1}^t = m_{[LOD_1,δ_t]}(ε_t)
19          eksekver list_1^p med α_{LOD_1}^p = α_{LOD_1}^t
20    ellers hvis ε_t ≤ LOD_2
21       eksekver list_0^t med α_{LOD_0}^t = 1.0
22       eksekver list_0^p med α_{LOD_0}^p = 1.0
23       eksekver list_1^t med α_{LOD_1}^t = 1.0
24       eksekver list_1^p med α_{LOD_1}^p = 1.0
25       hvis LOD_2 − trans ≥ δ_t
26          eksekver list_2^t med α_{LOD_2}^t = f_2(ε_t)
27          eksekver list_2^p med α_{LOD_2}^p = α_{LOD_2}^t
28       ellers
29          eksekver list_2^t med α_{LOD_2}^t = m_{[LOD_2,δ_t]}(ε_t)
30          eksekver list_2^p med α_{LOD_2}^p = α_{LOD_2}^t
```

Vi ser at vi undersøker for hver LOD_i trerepresentasjon om $LOD_i - \mathit{trans} \geq \delta_t$. Om dette er tilfelle kan vi blende med $f_i(\epsilon_t)$ siden denne garanterer at alfaverdien til LOD_i representasjonen vil være 1 når $\epsilon_t = LOD_i - \mathit{trans}$. Om derimot $LOD_i - \mathit{trans} < \delta_t$ må vi blende inn med $m_{[LOD_i,\delta_t]}(\epsilon_t)$ for å forsikre oss om at alfaverdien til LOD_i representasjonen er 1 når $\epsilon_t = \delta_t$. Siden en tile er som tilfredsstiller $\delta_p \geq LOD_0$ ikke har tegnet noen trær i foreldretilen, fordi LOD_0 skiftet forekommer først under traverseringen av t, må vi blende inn alle trærne i t med samme alfaverdi. Dette gjelder altså både de utplasserte trærne i t og de dupliserte treposisjonene fra foreldretilen p.

Vi er nå klare for å se på alle tiler t der $\delta_p < LOD_0$. Dette vil si at foreldretilen p allerede har blendet inn sine planrepresentasjoner slik at disse har alfaverdi 1 når $\epsilon_t = \delta_p$ for t. De fire tilfellene som da kan forekomme er vist på figur 6.12.



Figur 6.12: *De fire tilfellene vi må ta hensyn til dersom* $\delta_p < LOD_0$.

De to øverste tilfellene på figur 6.12 er like i den forstand at $\delta_t > LOD_1$; altså LOD_1 skiftet vil ikke forekomme i t. Vi må dermed sørge for å blende inn planrepresentasjonene som ble utplassert i t fra $\epsilon_t = \delta_p$. Siden planrepresentasjonene i foreldretilen p til t allerede har alfaverdi 1 når $\epsilon_t = \delta_p$ kan vi tegne $list_0^p$ med $\alpha^p_{LOD_0} = 1$.

Det er igjen verdien til blendinglengden *trans* som bestemmer hvilket tilfelle som vil forekomme. Vi må altså undersøke om $\delta_p - trans < \delta_t$ for tilen t. Hvis $\delta_p - trans \geq \delta_t$, som tilsvarer det øverste tilfellet på figur 6.12, kan vi velge $\alpha^t_{LOD_0} = g_{\delta_p}(\epsilon_t)$ som i ligning (6.14). Om $\delta_p - trans < \delta_t$ må blende inn fra $\epsilon_t = \delta_p$ til $\epsilon_t = \delta_t$. Vi må da velge $\alpha^t_{LOD_0} = m_{[\delta_p,\delta_t]}(\epsilon_t)$.

Tilsvarende prosess må gjøres for de to tilfellene nederst på figur 6.12. Her vil LOD_1 skiftet forekomme innenfor t og vi må dermed sørge for at $\alpha^t_{LOD_0} = 1$ når $\epsilon_t = LOD_1$. Dette må gjøres fordi vi krever at forrige trerepresentasjonene skal ha alfaverdi 1 når en ny trerepresentasjon skal tegnes.

Vi må nå undersøke hvorvidt $\delta_p - trans < LOD_1$. Er $\delta_p - trans \geq LOD_1$ kan vi velge $\alpha^t_{LOD_0} = g_{\delta_p}(\epsilon_t)$ som i ligning (6.14) akkurat som i det øverste tilfellet på figur 6.12. Er derimot $\delta_p - trans < LOD_1$, må vi velge $\alpha^t_{LOD_0} = m_{[\delta_p,LOD_1]}(\epsilon_t)$.

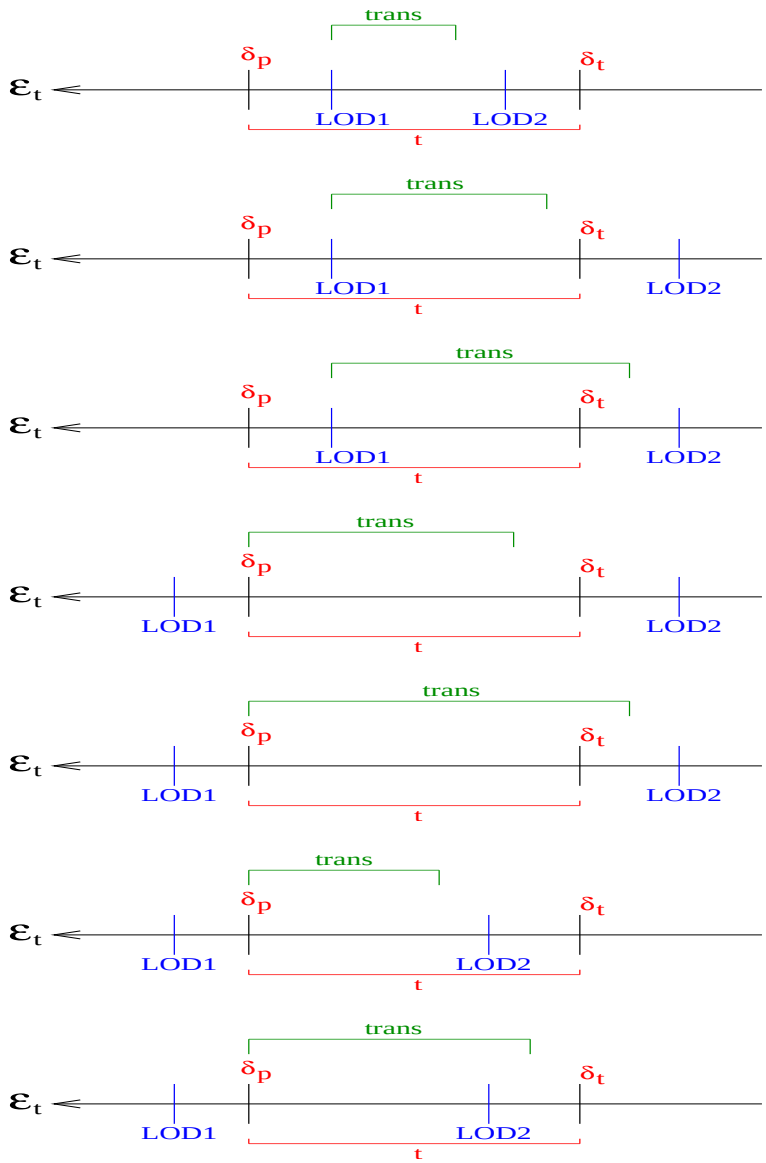
Vi er nå klare for å gi fortsettelsen på algoritme 6.5 for tilfellet hvor vi skal blende inn planrepresentasjonene for en tile t der $\delta_p < LOD_0$:

Algoritme 6.6, tegnMedDybderendering(t) [II]

```
31 ellers hvis δ_p < LOD_0
32     hvis ε_t > LOD_1
33         eksekver list_0^p med α^p_{LOD_0} = 1.0
34         hvis δ_t > LOD_1
35             hvis δ_p − trans ≥ δ_t
36                 eksekver list_0^t med α^t_{LOD_0} = g_{δ_p}(ε_t)
37             ellers
38                 eksekver list_0^t med α^t_{LOD_0} = m_{[δ_p, δ_t]}(ε_t)
39         ellers
40             hvis δ_p − trans ≥ LOD_1
41                 eksekver list_0^t med α^t_{LOD_0} = g_{δ_p}(ε_t)
42             ellers
43                 eksekver list_0^t med α^t_{LOD_0} = m_{[δ_p, LOD_1]}(ε_t)
```

Når vi skal blende inn kryss-planrepresentasjonene for en tile der $\delta_p < LOD_0$ er det enda flere spesialtilfeller som vi må ta hensyn til. Alle tilfellene som kan forekomme for en tile t er vist på figur 6.13. For de tre øverste tilfelle på figur 6.13 har vi at $\delta_p \geq LOD_1$. Dette vil si at LOD_1 skiftet forekommer under opptegningen av t. De resterende nederste fire tilfellene på figur 6.13 tilsvarer de fire tilfellene vi så på for planrepresentasjonen på figur 6.12. Siden blendingen her må ta samme hensyn som for planrepresentasjonene, er det unødvendig å gjenta disse. Vi konsentrerer oss derfor kun om de tre øverste tilfellene på figur 6.13 der LOD_1 skiftet forekommer under opptegningen av t.

Det øverste tilfellet på figur 6.13 er også det enkleste. Siden vi er garantert at $LOD_1 - trans > LOD_2$ blender vi inn med $f_1(\epsilon_t)$. De to resterende tilfellene tilsvarer figur 6.11 (ved å sette $LOD_0 = LOD_1$ og $LOD_1 = LOD_2$ i figuren). Hvis $LOD_1 - trans < \delta_t$ må vi blende inn med $\alpha^p_{LOD_1} = \alpha^t_{LOD_1} = m_{[LOD_1,\delta_t]}(\epsilon_t)$ som i ligning (6.23). Ellers blender vi inn med $f_1(\epsilon_t)$.



Figur 6.13: *Alle tilfellene som kan forekomme dersom vi skal tegne kryss-planrepresentasjonene i en tile t der $\delta_p < LOD_0$. De tre øverste figurene viser tilfellene der LOD_1 skiftet forekommer under opptegningen av t. Siden kryss-planrepresentasjonen ikke har vært blendet inn for hverken trærne i t eller de dupliserte treposisjonene til t, må vi sørge for at begge blir blendet inn samtidig. De fire nederste figurene tilsvarer de fire tilfellene for planrepresentasjonene på figur 6.12 som vi allerede har sett på.*

Vi er nå klare for å gi forsettelsen på *tegnMedDybdeRendering(...)* for tilfellet hvor $\delta_p < LOD_0$ for en tile t og $LOD_1 \geq \epsilon_t > LOD_2$. Med andre ord så ønsker vi å blende inn kryss-planrepresentasjonen:

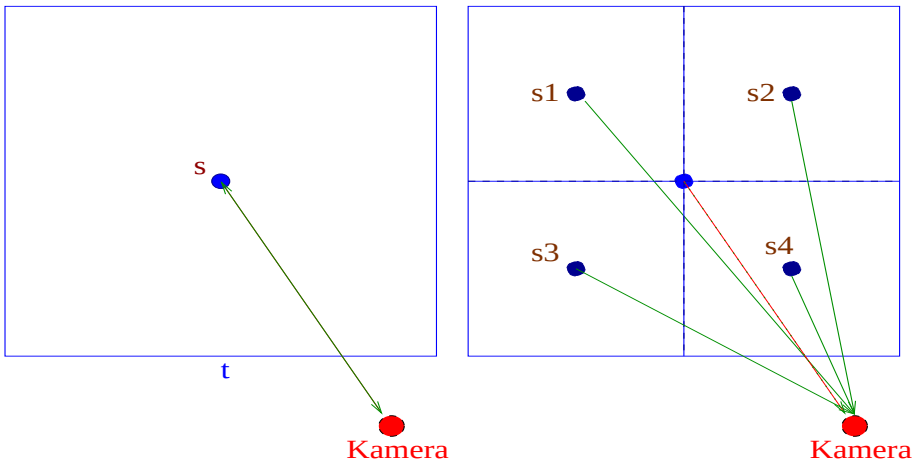
Algoritme 6.7, tegnMedDybderendering(t) [III]

```
44  ellers hvis LOD_1 ≥ ε_t > LOD_2
45      hvis δ_p ≥ LOD_1
46          eksekver list_0^t med α_{LOD_0}^t = 1
47          eksekver list_0^p med α_{LOD_0}^p = 1
48          hvis δ_t > LOD_2 og LOD_1 − trans < δ_t
49              eksekver list_1^t med α_{LOD_1}^t = m_{[LOD_1, δ_t]}(ε_t)
50              eksekver list_1^p med α_{LOD_1}^p = α_{LOD_1}^t
51          ellers
52              eksekver list_1^t med α_{LOD_1}^t = f_1(ε_t)
53              eksekver list_1^p med α_{LOD_1}^p = α_{LOD_1}^t
54      ellers
55          eksekver list_0^p med α_{LOD_0}^p = 1
56          eksekver list_1^p med α_{LOD_1}^p = 1
57          hvis δ_t > LOD_2
58              hvis δ_p − trans ≥ δ_t
59                  eksekver list_0^t med α_{LOD_0}^t = g_{δ_p}(ε_t)
60                  eksekver list_1^t med α_{LOD_1}^t = α_{LOD_0}^t
61              ellers
62                  eksekver list_0^t med α_{LOD_0}^t = m_{[δ_p, δ_t]}(ε_t)
63                  eksekver list_1^t med α_{LOD_1}^t = α_{LOD_0}^t
64          ellers
65              hvis δ_p − trans ≥ LOD_2
66                  eksekver list_0^t med α_{LOD_0}^t = g_{δ_p}(ε_t)
67                  eksekver list_1^t med α_{LOD_1}^t = α_{LOD_0}^t
68              ellers
69                  eksekver list_0^t med α_{LOD_0}^t = m_{[δ_p, LOD_2]}(ε_t)
70                  eksekver list_1^t med α_{LOD_1}^t = α_{LOD_0}^t
```

For blending med stjerne-planrepresentasjonene er det nøyaktig samme fremgangsmåte som for de andre LOD nivåene. Vi må passe oss for tilfellet hvor $\delta_p - trans < 0$. Dette tilsvarer samme tilfelle som for direkterendering. Vi lar hver å gi algoritmen for blending med stjerne-planrepresentasjonene.

6.3 Endring av avstandsmål

Vi har til nå sagt at vi traverserer ned til en tile t når $\epsilon_t < \delta_p$. Vi skal nå vise at dette ikke er riktig. Figur 6.14 illustrerer dette.



Figur 6.14: *La oss anta at vi har gitt tilen t til venstre på figuren. Her er $\epsilon_t = \delta_t$. Vi flytter så kamera litt mot t slik at $\epsilon_t < \delta_t$. Vi skal dermed traversere videre ned i barna til t som vist til høyre på figuren. Vi ser at avstanden til de fire barna til t varierer.*

For tilene (s_1, s_2) til høyre på figur 6.14 vil avstandsmålet $(\epsilon_{s_1}, \epsilon_{s_2})$ være større enn ϵ_t. Dette kommer av at avstanden fra midtpunktet i disse til kameraet er større enn avstanden fra midpunktet i t til kameraet. Dette vil si at for tilene (s_1, s_2) på figur 6.14 vil $(\epsilon_{s_1}, \epsilon_{s_2})$ være større enn ϵ_t verdien som gjorde at vi traverste ned i dem. Dette igjen tilsvarer at $(\epsilon_{s_1}, \epsilon_{s_2}) > \delta_p$ der δ_p er objektfeilen til t siden t er foreldretilen til tilene (s_1, s_2).

Under visualiseringen er ikke dette noe problem, da $(\epsilon_{s_1}, \epsilon_{s_2})$ vil bli δ_p etterhvert som vi nærmer oss (s_1, s_2). Siden vi for mange tilfeller under visualiseringen blender inn fra $\epsilon_t = \delta_p$, vil de ulike blendingsfunksjonene gi alfaverdier mindre enn 0 når $\epsilon_t > \delta_p$. OpenGL tolker alfaverdier mindre enn 0 som om alfaverdien var lik 0. Dermed vil ingen visuelle effekter forekomme i tiler der ϵ_t er større enn objektfeilen til foreldretilen.

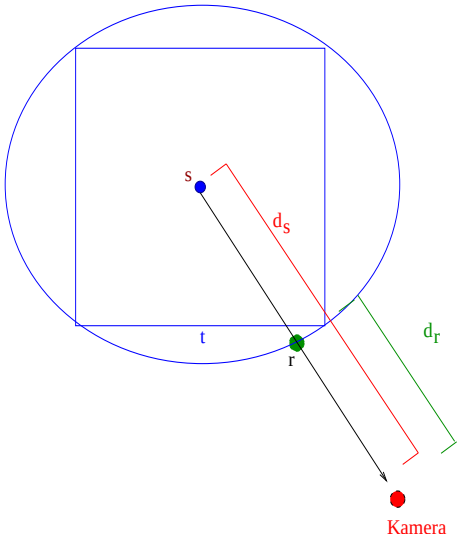
Verre er det for tilene (s_3, s_4) til høyre på figur 6.14. Her er avstanden fra midtpunktet i disse til kameraet mindre enn avstanden fra t. Vi får dermed at $(\epsilon_{s_3}, \epsilon_{s_4})$ vil bli mye mindre enn δ_p. Siden vi har antatt at vi traverserer ned til en tile med en gang $\epsilon_t < \delta_p$ vil vi få kunstige effekter under visualiseringen dersom dette ikke stemmer. Dette kommer av at mindre avstand gir større alfaverdier for våre blendingsfunksjoner. Om da avstandsmålet ϵ_t til en tile er mye mindre enn δ_p med en gang vi traverserer ned til en tile t, vil vi oppleve at trerepresentasjonene får en alfaverdi større enn 0. Dette fører til at trerepresentasjonene som skal blendes inn fra δ_p plutselig popper fram i landskapet.

For å løse dette problemet må vi endre vårt avstandsmål ϵ_t. Vi husker at vår originale ϵ_t mål er gitt ved:

$$\epsilon_t = \theta d_s, \tag{6.27}$$

hvor d_s er lengden til vektoren fra midtpunktet s i en tile t til det virtuelle kameraet og θ er vinkelen til en piksel på skjermen.

Om vi nå normaliserer vektoren fra s til kameraet og ganger denne med radien til den omringede sfæren til tilen t, får vi punktet r på sfæren slik som på figur 6.15. Tanken er nå at vi også ønsker å måle lengden til vektoren fra r til det virtuelle kameraet. Vi kaller denne avstanden for d_r.



Figur 6.15: *Vi introduserer det nye punktet r som ligger på den omringede sfæren til t. Avstanden fra r til det virtuelle kameraet kaller vi for d_r. Avstanden fra midtpunktet s til kameraet kaller vi for d_s.*

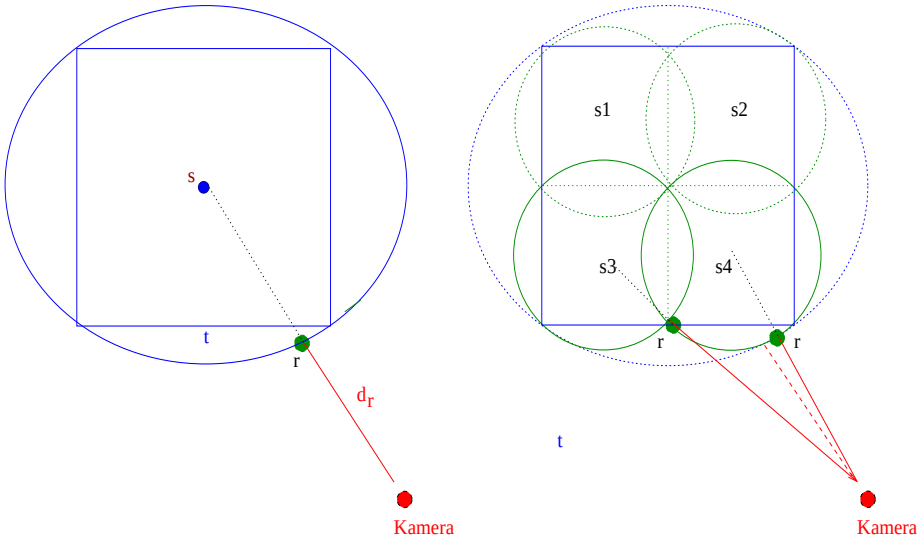
For hver tile t sjekker vi nå de to avstandsmålene d_s og d_r mot hverandre. Vi ønsker alltid å bruke den minste avstanden, slik at vi velger d_s hvis $d_s < d_r$. Ellers velger vi d_r.

Etter å ha funnet den korteste avstanden, multipliserer vi denne med θ som i ligning (6.27). Vi får dermed en ny ϵ_t verdi. Grunnen til dette er at vi ønsker nå minimerer avviket i avstandsmålene som kan forekomme når vi traverserer dypere ned i quadtreet.

Gitt nå en tile t som til venstre på figur 6.16. Her ser vi at lengden d_r fra r til det virtuelle kameraet er mindre enn lengden d_s fra midtpunktet s til t. Vår nye ϵ_t verdi blir dermed

$$\epsilon_t = \theta d_r. \tag{6.28}$$

Anta nå at vi videre skal traversere ned i barna til t. De fire barna (s_1, s_2, s_3, s_4) til høyre på figur 6.16 tilsvarer de samme barna som til høyre på figur 6.14. Fortsatt vil avstandsmålet $(\epsilon_{s1}, \epsilon_{s2})$ være større enn avstandsmålet til foreldretilen t. Som før har ikke dette noe å si under visualiseringen.



Figur 6.16: *Vi introduserer det nye punktet r som ligger på den omringede sfæren til t. Avstanden fra r til det virtuelle kameraet kaller vi for d_r. Avstanden fra midtpunktet s til kameraet kaller vi for d_s.*

Vi konsentrerer oss derfor om tilene (s_3, s_4) til høyre på figur 6.16. For begge disse begynner vi med å finne punktet r på sfæren. Begge disse er markert med grønne punkter på figuren. Vi ser videre at vi for både s_3 og s_4 finner at avstanden fra sine respektive r til kameraet er mindre enn avstanden fra midtpunktene s til kameraet. Vi velger dermed d_r for begge disse tilene og multipliserer disse med θ for å få nye avstandsmål $(\epsilon_{s_3}, \epsilon_{s_4})$.

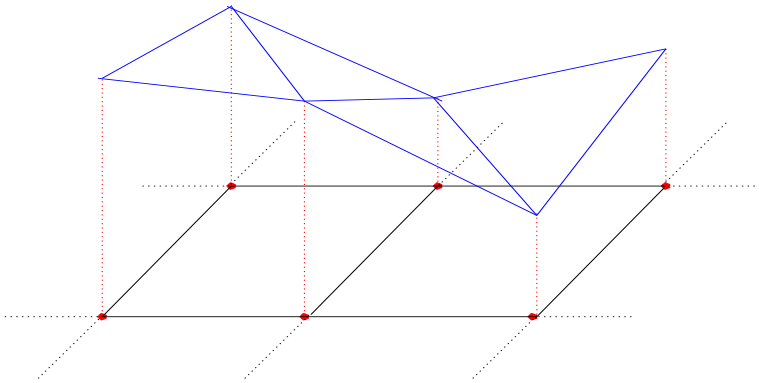
Det som vi nå legger merke til fra figur 6.16 er at forskjellen fra foreldretilen sitt avstandsmål og barnetilene sine avstandsmål blir betydelig minsket. Den stiplede rød linjen til høyre på figur 6.16 representerer avstanden fra foreldretilen t sin r og det virtuelle kameraet. De to heltrukne rød strekene gir de to avstanden fra henholdsvis s_3 og s_4 sin r og kameraet. Vi ser at forskjellen mellom disse er betydelig mindre enn til høyre på figur 6.14.

Ut i fra dette kan vi konkludere med at dette nye avstandsmålet er hensiktsmessig for vårt formål. For hver tile t under traverseringen må vi altså i tillegg til å beregne avstanden d_s fra midtpunktet i t til kameraet også beregne punktet r og avstanden d_r fra denne til kameraet. Så velger vi minste avstanden d_s eller d_r og multipliserer denne med θ for å finne det nye avstandsmålet ϵ_t.

6.4 Høydeverdier

Vi skal nå se på en forenklet modell for å tilføre høydeverdier i modellen vår. Denne modellen er kun gitt for å illustrere hvilken hensyn vi må ta dersom vi ønsker å tilføre høydeverdier til hvert tre i skogmodellen.

Vi begynner denne diskusjonen ved å se på hvordan vi kan knytte høydeverdier til rasteret vårt. Det er flere måter å gjøre dette på: vi kan enten forbinde hver celle i rasteret med en høydeverdi eller gi fire høydeverdier for hjørnene i en celle. For vårt tilfelle har vi valgt sistnevnte fremgangsmåte. Gitt et raster på N rader og kolonner må vi dermed ha $N + 1$ høydeverdier i begge retninger.

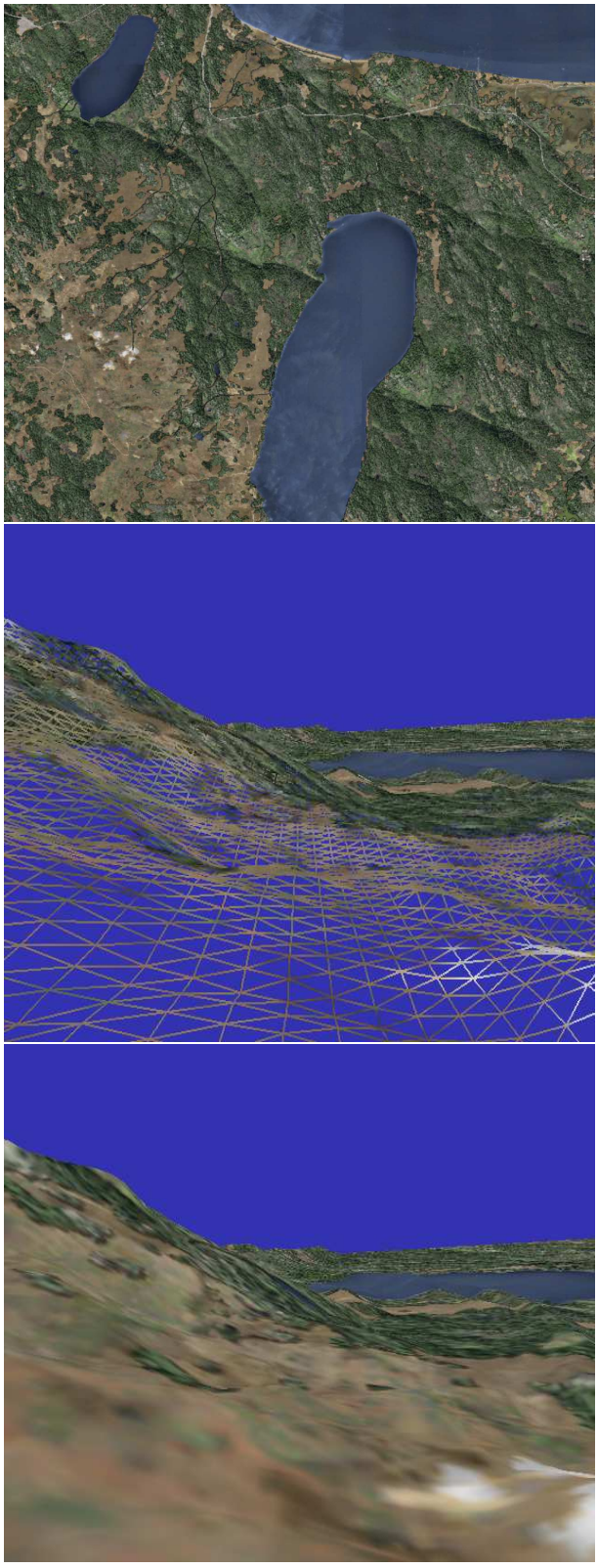


Figur 6.17: *Figuren viser hvordan vi kan assosiere høydeverdier til hver celle i rasteret. Hver celle har gitt høydeverdier i sine fire respektive hjørner og vi forbinder hver celle med to trekanter.*

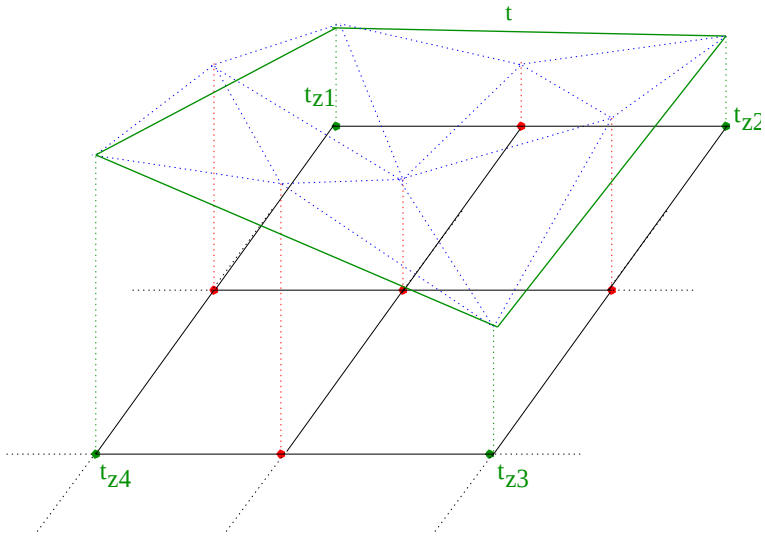
Vi knytter nå hver celle i rasteret med *to* trekanter som på figur 6.17. Hver rad i rasteret kan dermed tegnes som en "triangle strip" for å redusere geometrien som må behandles av grafikkakseleratoren. For å øke realismen teksturerer vi landskapet med en tekstur som er satt sammen av ulike satellittfotografier. Figur 6.18 viser denne nye teksturen sammen med resultatet av å teksturere trekantene.

Etter at vi har innført høydeverdier for rasteret vårt må vi innføre høydeverdier for hver tile i quadtreet. Siden hver tile dekker over et visst antall celler i rasteret kan vi utnytte høydeverdiene til disse for å finne høydeverdiene til en tile. Et eksempel på hvordan vi tilordnet høydeverdier til en tile som spenner ut fire celler i rasteret er vist på figur 6.19.

Ved å utnytte denne fremgangsmåten vil en tile automatisk "forme" seg etter landskapet. For tiler på de øverste nivåene i quadtreet som spenner over store områder i terrenget vil nødvendigvis utformingen være grov. Ved traversering dypere ned i quadtreet vil en tile dekke stadig mindre områder i terrenget. Dermed blir utformingen av landskapet for quadtreet også finere og finere.

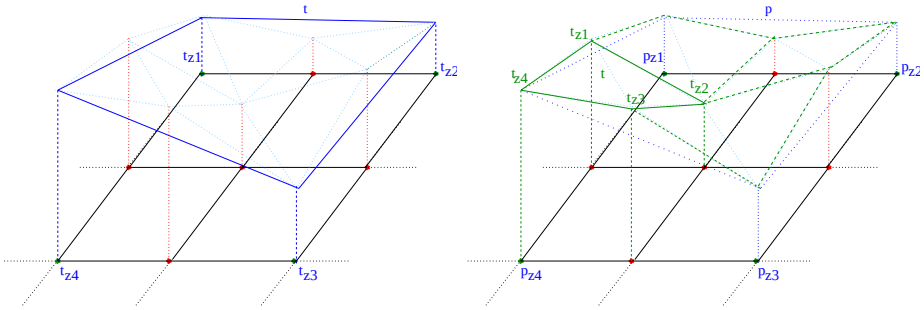


Figur 6.18: *Vi teksturerer landskapet med en tekstur av samme området som er dekket av rasteret. Denne teksturen er satt sammen av satellittfotografier og er vist øverst. Resultatet av å gi høydeverdier for rasteret kan sees på de to nederste figurene. I midten har vi kun tegnet trekantene.*



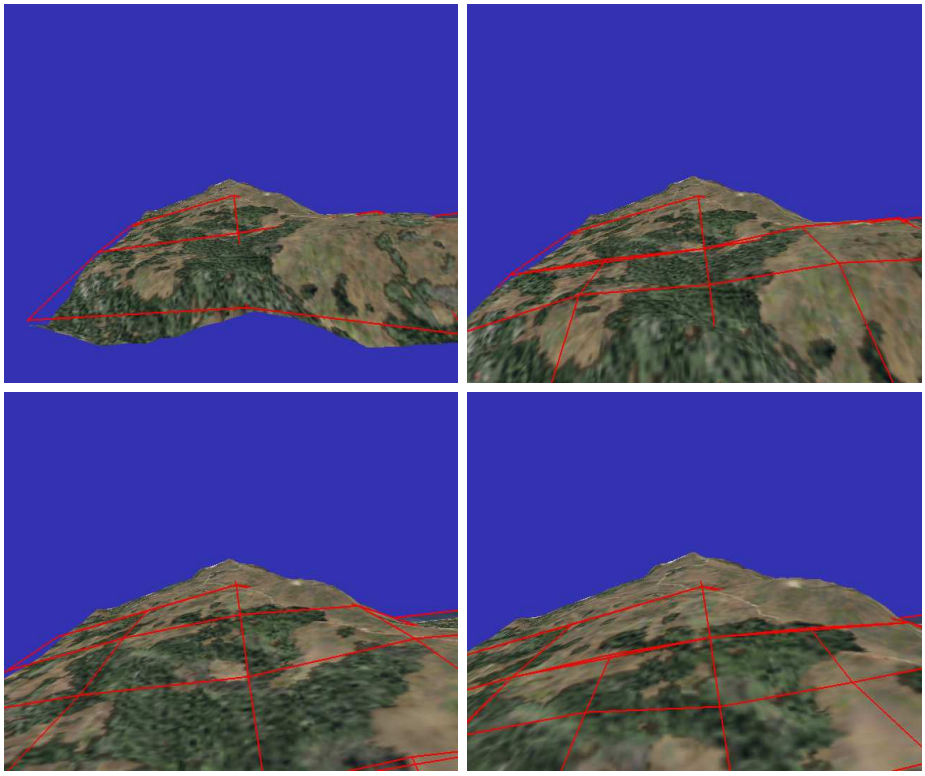
Figur 6.19: *Figuren viser hvordan vi forbinder de fire hjørnene i en tile med de høydeverdiene i det underliggene rasteret. Vi innfører* $(t_{z1}, t_{z2}, t_{z3}, t_{z4})$ *for høydeverdiene til en tile* t*.*

Traversering ned i barna til en tile t vil føre til at at vi får "sprekker" i quadtreet. Dette er vist på figur 6.20. For terrengvisualisering er dette et problem som må tas hensyn til siden hver tile i quadtreet representerer geometrien i landskapet [8]. For vårt formål derimot representerer en tile kun statiske treposisjoner, så vi slipper å ta hensyn til denne problemstillingen.



Figur 6.20: *Ved å gi høydeverdiene i en tile ut i fra rasteret vil vi oppleve at instansieringen av barn fører til sprekker i quadtreet. Til venstre har vi en tile* t *som vi ønsker å forbinde med fire barn vist til høyre.*

Figur 6.21 viser dette i praksis for rasterdatasettet vårt. Her ser vi at jo mindre avstanden er fra det virtuelle kameraet til landskapet, jo dyperer ned i quadtreet må vi traversere. Dette fører til at tiler i quadtreet former seg mer og mer etter terrenget. De to nederste figurene på figur 6.21 viser også tydelig sprekkene som forekomme for en tile i quadtreet.



Figur 6.21: *Eksempel på hvordan quadtreet former seg etter landskapet.*

For å traversere quadtreet husker vi fra seksjon 6.3 at vi måtte ha et avstandsmål ϵ_t som var avhengig av avstanden fra midtpunktet s i en tile til kameraet. For å finne z-verdien til s tar vi gjennomsnittet av de fire z-verdiene for t ved:

$$s_z = \frac{1}{4}(t_{z1} + t_{z2} + t_{z3} + t_{z4}), \tag{6.29}$$

hvor t_{zi} er gitt som på figur 6.19.

For å beholde våre omringede sfærer gir vi radien rad til denne ved:

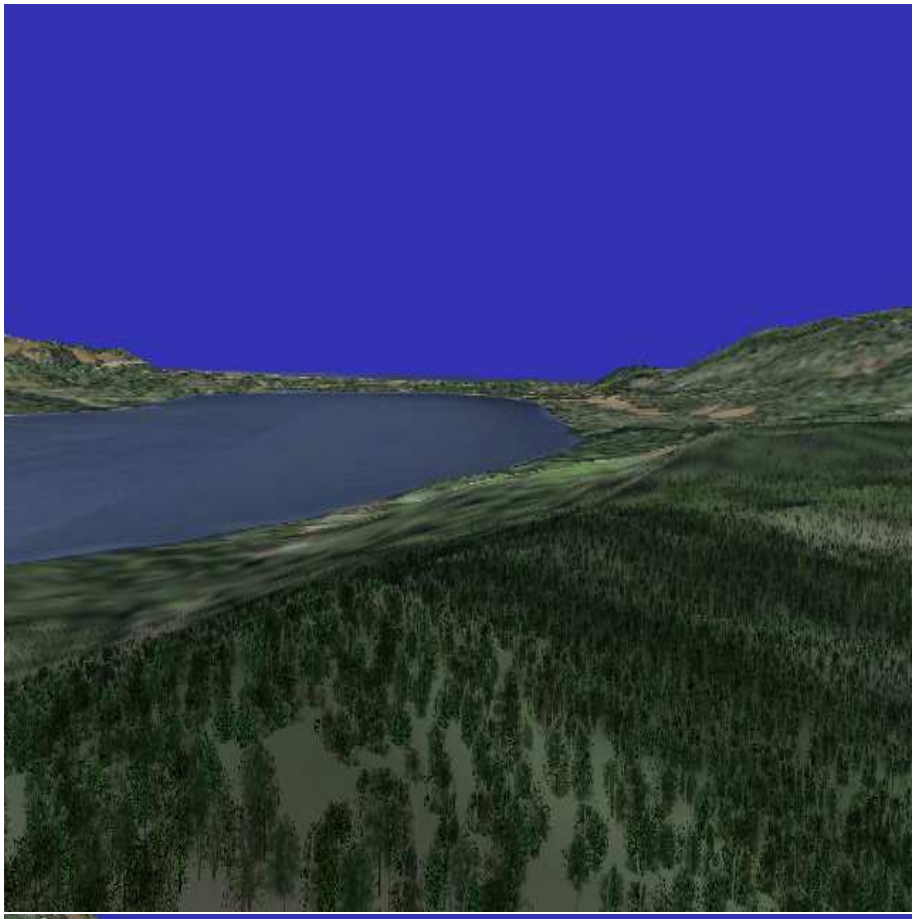
$$rad = |(t_{x_{maks}}, t_{y_{maks}}, t_{z2}) - (s_x, s_y, s_z)|. \tag{6.30}$$

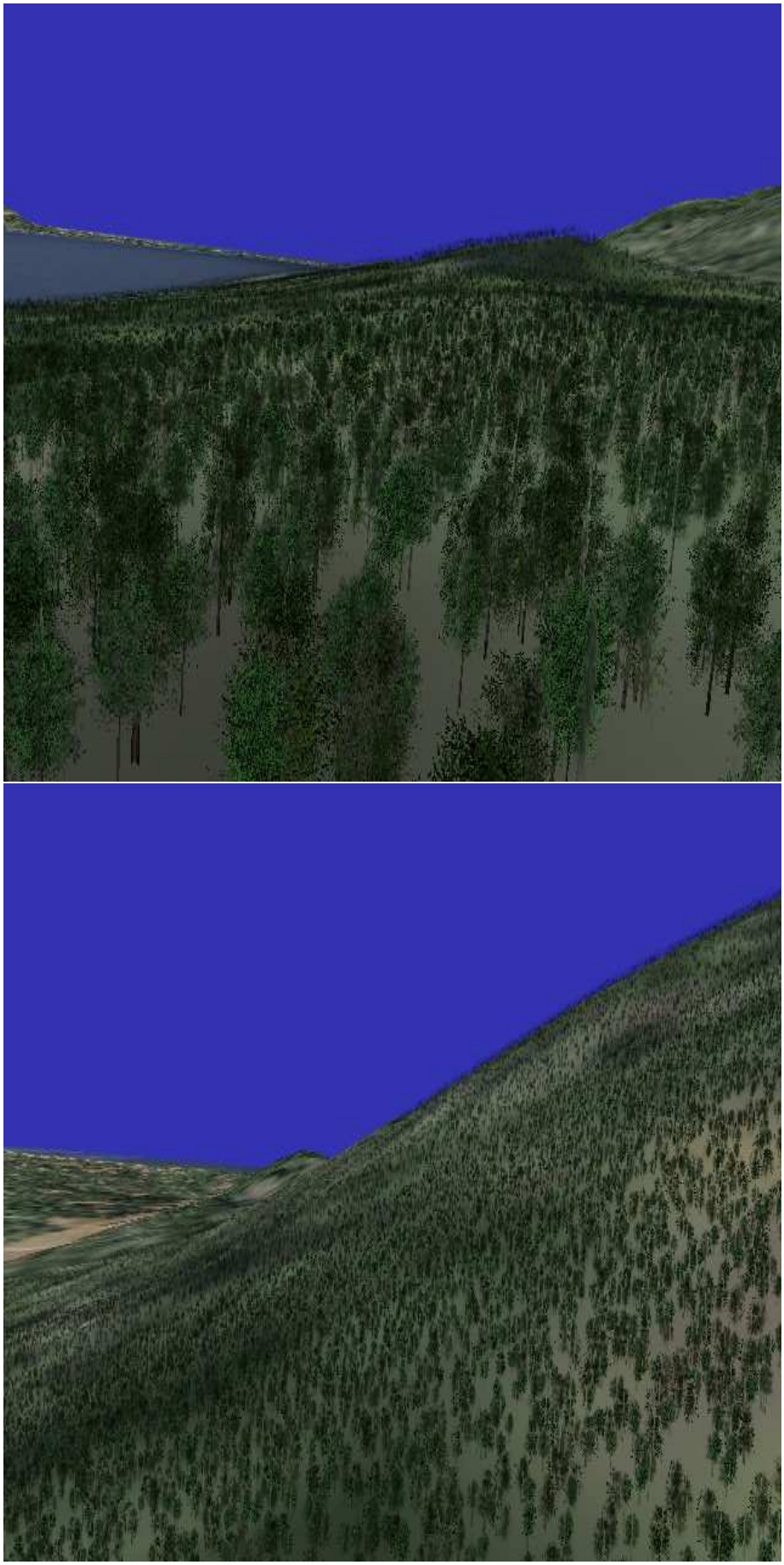
Merk at vi både kan få tilfeller hvor sfæren ikke lenger vil være inneholdt i foreldresfæren, og tilfeller hvor sfæren ikke inneslutter hele geometrien i terrenget. Dette faller derimot utenfor denne oppgaven.

Til slutt må vi beregne z-verdien til alle treposisjonene innenfor en tile t. Vi kan forenkle dette fordi vi vet hvilken rastercelle en treposisjon ligger innenfor. Dermed må vi beregne treposisjonen sin høyde ut i fra de fire høydeverdiene til en celle i rasteret. Dette kan for eksempel gjøres ved bilineær interpolasjon.

Grunnen til at denne høydemodellen ikke er realistisk er at den ikke tar hensyn til om deler av terrenget er nærme eller langt unna. Hele landskapet blir dermed tegnet med samme detaljgrad. Det er derimot ønskelig å tilføre et LOD rammeverk også for terrengvisualisering. Hensyn som må tas for å implementere skogsmodellen i et slikt rammeverk er diskutert under videre arbeid i seksjon 8.2.

Vi gir til slutt det visuelle resultatet av å gi høydeverdier til trær i et terreng. Her tegner vi omtrent 60000 trær per bilderamme. Vi legger til disse skjermbildene for å vise hva som er mulig med vår implementasjon.





Kapittel 7

Resultater

Vi er nå klare for å måle hvor mange trær vi klarer å visualisere i sanntid. Det viktig å merke seg at målingene er svært avhengig av hastigheten til grafikkakseleratoren. Hvor mange grafiske primitiver grafikkakseleratoren klarer å prosessere i sanntid vil ha innvirkning på hvor mange trær vi kan tegne til skjerm. For å måle resultatet av opptegningen har vi kjørt samme testdata på to ulike datamaskiner med to ulike grafikkakseleratorer.

7.1 Resultater

Som nevnt i innledningen er målet vårt å beholde kravet om sanntidsvisualisering. I denne oppgaven tilsvarer dette at en opptegningshastighet på 60 bilderammer per sekund er det minste vi kan tolerere under sanntidsvisualisering. Er opptegningshastigheten langt under dette, vil vi oppleve merkbare effekter under visualiseringen: forsinket prosessering av input fra tastatur eller mus og hakkete bevegelser når vi flytter det virtuelle kameraet.

Hvor mange grafiske primitiver en grafikkakselerator klarer å prosessere i sanntid vil ha innvirkning på hvor mange trær vi kan tegne til skjerm. Vi ønsker å måle antall trær med ulike typer grafikkakseleratorer for å få oversikt over begrensningene til vår modell. I denne oppgaven har vi målt resultatet med to ulike grafikkakseleratorer.

Vi begynner med å se på spesifikasjonene til de to datamaskinene som vi har brukt i denne oppgaven. Begge maskiner kjører "Debian" distribusjonen av Linux operativsystemet. Den første testmaskinen har følgende maskinvare:

CPU	Minne	Grafikkakselerator
Intel(R) Pentium(R) 4 CPU 1.70GHz	512 MB	GeForce2 MX/PCI/SSE2

Vi kaller denne maskinen for *testmaskin1*. Den andre maskinen vi testet skogmodellen på har følgende maskinvare:

CPU	Minne	Grafikkakselerator
AMD Athlon(TM) MP 1800+	1024 MB	GeForce4 Ti 4200/PCI/SSE/3DNOW!

Denne kaller vi for *testmaskin2*.

Vi har testet skogmodellen med tre ulike skogkonfigurasjonsfiler: *konf1*, *konf2* og *konf3*. Alle filene er gitt de samme trærne som i tillegg A. De ulike parametrene som er av spesiell interesse er gitt i følgende tabell:

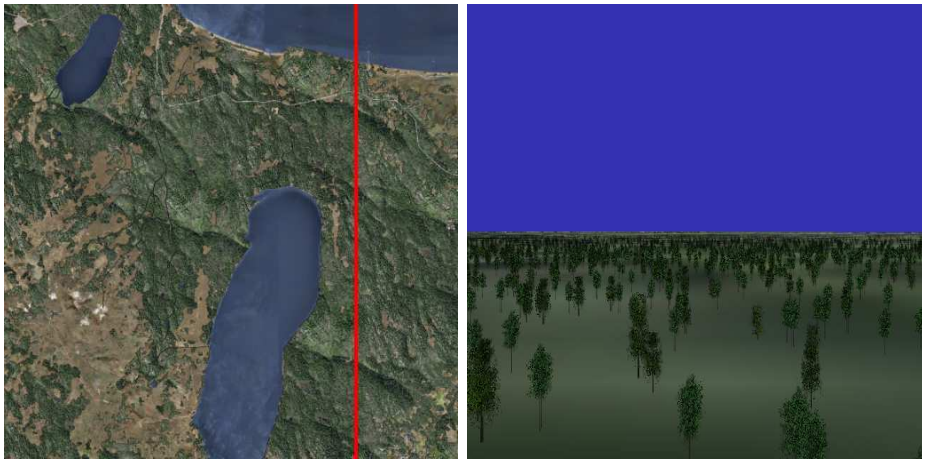
	konf1	*konf2*	*konf3*
η	500	750	1000
l_{maks}	7	7	7
M_{maks}	3	5	6
LOD_0	50	40	30
LOD_1	20	20	20
LOD_2	10	10	20
trans	9	9	9

Det viktigste å merke seg er at η verdien i de ulike skogkonfigurasjonsfilene øker fra 500 i *konf1* filen til 1000 i *konf3*. Dette tilsvarer at *konf3* filen utplasserer flest treposisjoner i landskapet. Siden flere treposisjoner fører til at mer maskinminne blir brukt til å lagre posisjonene, øker vi M_{maks} i hver skogkonfigurasjonsfil for å sørge for at minneopprydningsalgoritmen vår slår til like mange ganger for alle målingene våre. Vi senker også LOD_0 parameteren for hver skogkonfigurasjonsfil for å forhindre at for mange trær blir tegnet til skjerm på en gang.

Det er viktig å måle resultatet ved samme *kamerabane*. Vi definerer altså en fast bane som det virtuelle kamera skal forflytte seg i. Dette garanterer at målingene vi gjør foregår under de samme betingelsene. For vårt formål ønsker vi å måle ytelsen av skogmodellen i et planart landskap. Dette gjøres fordi høydemodellen introdusert i forrige kapittel ikke tar hensyn til "level of detail" metodikk og gir derfor en urealistisk måling av skogmodellen dersom vi hadde brukt denne.

Kamerabanen vi har brukt beveger seg vertikalt over det planare quadtreet. I tillegg plasserer vi kameraet i samme høyde over landskapet. Denne høyden er nærme nok landskapet til at vi traverserer ned til det nederste nivået l_{maks} i quadtreet og tegner den mest detaljerte trerepresentasjonen under gjennomkjøringen av kamerabanen. Figur 7.1 viser kamerabanen og et tilhørende skjermbilde fra målingene med *konf3* filen.

Det første vi ønsker å måle er gjennomsnittlig antall biledrammer per sekund under gjennomkjøringen av kamerabanen *uten* å tegne trær. Vi ønsker nemlig å undersøke hvor lang tid skogmodellen bruker på å instansiere nye tiler, generere treposisjoner og (eventuelt) duplisere treposisjoner. Målingene ble kun gjort for *konf3* skogkonfigurasjonsfilen, da dette gir "worst case"



Figur 7.1: *Til venstre er den vertikale kamerabanen vi har brukt for å måle skogmodellen markert med rødt. Området som er synlig for det virtuelle kameraet under gjennomkjøringen av kamerabanen er dekket av* $\approx 78\%$ *skogceller i det underliggende rasteret. Til høyre ser man et skjermbilde fra gjennomkjøringen av kamerabanen med konf3 skogkonfigurasjonsfilen.*

(flest treposisjoner) i vårt tilfelle. Resultatet av målingene av gjennomsnittlig antall bilderammer per sekund er gitt ved følgende tabell:

Gjennomsnittlig antall bilderammer per sekund		
Visualiseringsmetode	*testmaskin1*	*testmaskin2*
Direkterendring	536	766
Dybderendering	515	764

Det vi ser er at instansieringen av en tile og utplasseringen av treposisjoner er en rask prosess. Som ventet er direkterenderingsmetoden raskere enn dybderendering fordi man slipper å duplisere treposisjoner fra en foreldretile under instansiering. Forskjellen i antall gjennomsnittlig bilderammer per sekund er kun $536 - 512 = 24$ for kjøringen under *testmaskin1* og $766 - 764 = 2$ for kjøringen under *testmaskin2*.

Vi er nå klare for å måle gjennomsnittlig antall bilderammer per sekund ved opptegningen av våre trerepresentasjoner. Som over måler vi både dybde- og direkterendering. Resultatet av dette ved kjøring av *konf1* skogkonfigurasjonsfilen er gitt ved:

Gjennomsnittlig antall bilderammer per sekund		
Visualiseringsmetode	*testmaskin1*	*testmaskin2*
Direkterendering	33	132
Dybderendering	84	193

Her ser vi at vi får stor forskjell mellom gjennomsnittlig antall bilderammer per sekund for direkte- og dybderendering ($84 - 33 = 51$ bilderammer

for *testmaskin1*, og $193 - 132 = 61$ bilderammer for *testmaskin2*). For å forklare hvorfor forskjellen er så stor ser vi på gjennomsnittlig antall trær som skogmodellen forsøkte å tegne i hver bilderamme under samme gjennomkjøringen av kamerabanen. Resultatet av denne målingen er gitt ved:

Gjennomsnittlig antall trær per bilderamme		
Visualiseringsmetode	*testmaskin1*	*testmaskin2*
Direkterendering	5595	5602
Dybderendering	2886	2887

På *testmaskin1* forsøker altså skogmodellen å gjennomsnittlig tegne $5595 - 2886 = 2709$ flere trær i hver bilderamme under direkterendering, mens dette tallet for *testmaskin2* ga $5602 - 2887 = 2715$. Siden våre trerepresentasjoner består av trekanter, og det er disse grafikkakseleratoren jobber med, ser vi også på gjennomsnittlig antall trekanter som ble sendt videre til grafikkortet for prosessering for hver bilderamme. Dette er gitt ved følgende tabell:

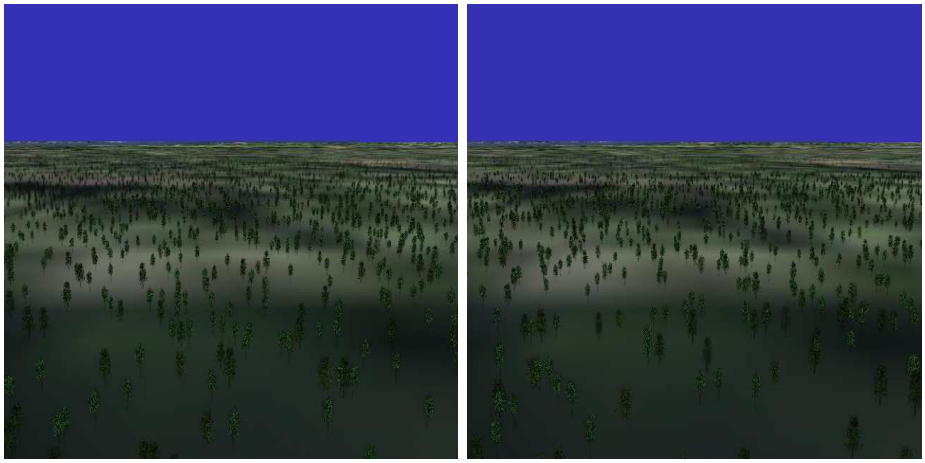
Gjennomsnittlig antall trekanter per bilderamme		
Visualiseringsmetode	*testmaskin1*	*testmaskin2*
Direkterendering	25183	25227
Dybderendering	9603	9610

Vi ser at forskjellen i våre to visualiseringsmetoder er enda større for antall trekanter: gjennomsnittlig $25183 - 9603 = 15580$ flere trekanter ble sendt til grafikkakseleratoren per bilderamme for direkterenderingsmetoden på *testmaskin1*, mens dette tallet var $25227 - 9610 = 15617$ for *testmaskin2*.

Grunnen til at forskjellen er så stor mellom direkte- og dybderendering er at direkterenderingsmetoden prøver å tegne alt for mange trær som ikke er synlige innenfor synsvolumet. Siden vi tegner alle trærne i en tile før traversering under direkterendering vil vi oppleve at tiler på de øverste nivåene i quadtreet vil dekke over et større område enn området innenfor synsvolumet. Dette fører til at mange trær havner utenfor synsvolumet til det virtuelle kameraet når vi prøver å tegne alle trærne til en slik tile. Grafikkakseleratoren må da prosessere flere grafiske primitiver og opptegningshastigheten vil dermed gå sakte.

Det er viktig å merke seg at direkterenderingsmetoden ikke fører til at flere trær blir synlig innenfor synsvolumet. Figur 7.2 viser det visuelle resultatet av en kjøring med samme skogkonfigurasjonsfilen for både dybde- og direkterendering. Vi ser at det er liten visuell forskjell mellom de to visualiseringsmetodene: direkterendering fører bare til at flere trær utenfor synsvolumet blir forsøkt tegnet til skjerm.

Vi kan dermed konkludere med at direkterendering er en dårlig visualiseringsmetode for vårt formål. Dybderendering gir bedre resultatet fordi vi venter med å tegne trær til vi har traversert til det nederste nivået i quadtreet. Dette fører til at dybderendering tegner trær i mindre områder i



Figur 7.2: *Til høyre ser vi resultatet av visualisering med dybderenderingsmetoden; venstre viser direkterendering. Siden vi genererer vilkårlig valgte treposisjoner vil det visuelle resultatet alltid variere fra en kjøring til en annen.*

landskapet, noe som igjen fører til at færre trær som ikke er synlige vil bli prosessert av grafikkakseleratoren. Vi ønsker derfor å kun konsentrere oss om dybderendering som visualiseringsmetode for skogmodellen, da vi har sett at denne metoden gir bedre resultat under visualiseringen.

Vi ønsker nå å måle resultatet for *konf2* skogkonfigurasjonsfilen vår. Vi har her økt η til 750 som vil føre til at flere treposisjoner blir utplassert i landskapet. Vi har samtidig senket LOD_0 slik at trærne tegnes ved et senere tidspunkt. Som nevnt over ser vi kun på resultatet av dybderenderingsmetoden:

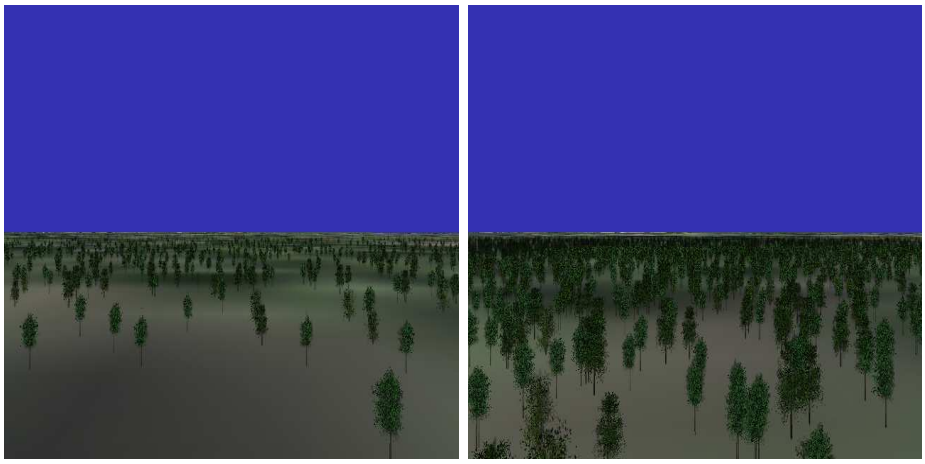
konf2		
mål	*testmaskin1*	*testmaskin2*
Gjennomsnittlig antall bilderammer per sekund	31	139
Gjennomsnittlig antall trær per bilderamme	3180	3181
Gjennomsnittlig antall trekanter per bilderamme	12110	12111

Her ser vi klart hvor mye grafikkakseleratoren har å si på resultatet: *testmaskin2* har en grafikkakselerator som er mye raskere enn *testmaskin1*. Vi ser av tabellen over at *testmaskin2* klarer å prosessere langt flere grafiske primitiver enn *testmaskin1* uten å komme under kravet vårt om 60 bilderammer per sekund. Denne forskjellen er enda mer tydelig for *konf3* filen. Her har vi igjen økt η til 1000 og senket LOD_0 ytterligere. Resultatene vi fikk er gitt ved:

konf3		
mål	*testmaskin1*	*testmaskin2*
Gjennomsnittlig antall bilderammer per sekund	22	111
Gjennomsnittlig antall trær per bilderamme	3423	3423
Gjennomsnittlig antall trekanter per bilderamme	14149	14141

Her ser vi at mens *testmaskin1* sliter med å prosessere 14149 trekanter i sanntid, har *testmaskin2* ingen problemer med denne datamengden. *testmaskin2* klarer faktisk 28281 trekanter per bilderamme og allikevel holde seg over 60 bilderammer per sekund, mens tilsvarende resultatet for *testmaskin1* er kun *9881* trekanter. Dette er en forskjell på 18400 trekanter per bilderamme for våre to testmaskiner.

Det er da klart at den visuelle forskjellen, målt i hvor mange trær vi klarer å tegne, vil være svært stor. Dette resultatet er vist på figur 7.3. Vi ser tydelig at jo flere trær vi klarer å utplassere i landskapet, jo bedre blir illusjonen av en ekte skog.



Figur 7.3: *Forskjell mellom antall trær våre ulike testmaskiner klarer å tegne uten å komme under vårt krav om* 60 *bilderammer per sekund. Til venstre ser vi resultatet med testmaskin1. Her tegner vi gjennomsnittlig 3493 trær per bilderamme. For testmaskin2 til høyre klarer vi å øke dette til 6845 trær. Det visuelle resultatet til høyre gir en langt bedre representasjon av en skog.*

Kapittel 8

Konklusjon

8.1 Konklusjon

Vi har sett at antall trær skogmodellen vår klarer å tegne til skjerm i sanntid er avhengig av grafikkakseleratoren. Hastigheten til slike kort fortsetter å øke fortløpende og det er i dag begrensningene på båndbredden mellom grafikkortet og cpu som er den største flaskehalsen i grafikksystemet. Realismen i vår modell er svært avhengig av hvor mange trær vi klarer å tegne til skjerm. Det er fortsatt langt igjen til vi klarer å tegne flere hundre tusener av trær til skjerm, men det er liten tvil om at hvis utviklingen av nye grafikkakseleratorer fortsetter i samme tempo vil vi nærme oss dette målet med stormskritt for hvert eneste år.

8.2 Videre arbeid

Til slutt vil vi konkludere med å se på mulig videre arbeid for skogmodellen vår. Det finnes en del arbeid både på integrering av skogmodellen i en terrengmodell, forbedring av trerepresentasjonene og multitråding som bør undersøkes videre. Vi gir her en kort innføring i hva som kan være aktuelle oppgaver for videre arbeid.

8.2.1 Integrering av skogmodellen i en terrengmodell

Høydemodellen introdusert i seksjon 6.4 tok ikke hensyn til ulik detaljgrad i landskapet. Akkurat som vi introduserte "level of detail" metoder for å representere trær og nivåer i quadtreet avhengig av avstanden til kameraet, bruker terrengmodeller de samme teknikkene for å effektivt tegne store terrengflater i sanntid. Slike terrengmodeller bruker ofte et quadtre som basis for visualiseringen, men hver tile er da forbundet med geometrien i landskapet. Traversering videre ned i quadtreet gir da finere oppløsning av terrenget [5], [6], [7], [8].

For å integrere skogmodellen i en slik terrengmodell må feilmålene i skogmodellen være assosiert med feilmålene i terrengmodellen. Det kan være ønskelig å forbinde de nederste nivåene i skogmodellen med de nederste nivåene i terrengmodellen siden trær ikke tegnes før man er nærme landskapet. Siden terrengmodeller generelt har mer kompliserte kriterier for hvilken detaljrepresentasjon som skal velges bør vi bytte ut objektfeilene for hver tile i vår modell med feilmålene til terrengmodellen.

I en terrengmodell representerer en tile geometrien til landskapet ved en gitt detaljgrad. Denne geometrien kan består av mange trekanter. Hvis vi har utplassert en treposisjon i en rastercelle, må vi nå finne hvilken trekant i terrengmodellen innenfor samme tile som treposisjonen hører til. Dette kan tenkes løst ved å projisere trekantene ned på planet som tilen spenner over, finne trekanten og beregne høydeverdien innenfor denne.

Videre har vi problematikken ved at finere oppløsning av terrenget fører til endring av geometrien. Dermed vil ikke høydeverdien til et tre for en detaljgrad være statisk men hele tiden endre seg ettersom vi forfiner terrenget. Uten å ta hensyn til dette vil da oppleve at trær i verste fall enten henger i løse luften eller blir begravd i landskapet. Dybderenderingsmetoden vår kan utnyttes til å løse denne problemstillingen på følgende måte: for hver gang vi må duplisere en treposisjon fra en foreldretile, beregner vi den nye høydeverdien for treposisjonen i den mer detaljerte terrengrepresentasjonen. Vi kan eventuelt bruke blending for å forhindre at trærne "hopper" i landskapet når de må tegnes i en ny høyde.

Flere områder som må tas hensyn til i en terrengmodell er mer kompliserte synlighetsspørringer. Dette kan videreføres direkte til vår skogmodell: er trær gjemt bak et fjell, ønsker vi heller ikke å tegne disse. En mulig løsning på dette problemet kan være å *gruppere* flere trær innenfor en tile for så å sjekke synligheten til hver gruppering i en tile.

8.2.2 Multitråder

Vi nevnte i seksjon 4.6 at opprydningsfunksjonen som blir kalt når vi har nådd den maksimale minnetoleransen vår fører til at applikasjonen midlertidig stopper opp. Samtidig kan vi regne med at om vi ønsker å integrere skogmodellen vår med en terrengmodell, vil beregningene av høydeverdiene til hver enkelt treposisjon ta tid. I sanntidsvisualisering er slike effekter i en applikasjonen spesielt merkbare fordi antall bilderammer per sekund blir betraktelig mindre så lenge slike beregninger pågår.

For å sørge for god flyt i applikasjonen bør vi implementere et *multitråd* rammeverk. En *tråd* er ansvarlig for en prosess i en applikasjon. Vi ønsker å ha *en* tråd som er kun ansvarlig for opptegningen, og *en* tråd som er kun ansvarlig for å instansiere og slette tiler i skogmodellen.

Implementasjonen bør være slik at instansieringen av nye tiler foregår *før* de eventuelt skal tegnes. Dette kan tenkes løst ved at tråden ansvarlig for å

instansiere tiler gjør dette når avstandsmålet $\epsilon_t = \xi_n - \Delta$ for en tile t. Siden vi bytter til barna til t når $\epsilon_t = \xi_n$ vil barna allerede være tilgjengelig under traversering dersom Δ er stor nok. Ved å kun ha en tråd som er ansvarlig for opptegning kan denne hele tiden anta at den andre tråden sørger for å ha all informasjon tilgjengelig som opptegningen bruker til enhver tid. Dermed slipper vi blant annet å undersøke om en tile er instansiert under traversering.

Ulempen med et multitråd rammeverk er at implementasjonen blir mer komplisert. Problemstillinger som hvor ofte en tråd skal eksekvere sine funksjoner, hvilke variable som må beskyttes mot overskriving og hvordan trådene skal dele informasjon er alle viktige områder som må tas hensyn til. Andre områder som må undersøkes er om opprydningen av tiler i quadtreet kan gjøres oftere, eventuelt om man kan spare minne ved å slette treposisjoner i en foreldretile som har vært duplisert lenger ned i quadtreet. Om vi velger å gjøre dette må vi ha et rammeverk for å kopiere de dupliserte treposisjonene til en foreldretile igjen.

8.2.3 Trerepresentasjoner

Trerepresentasjonene våre gir dårlig visuelt resultatet om vi:

- *Ser rett ned på trærne*: Siden våre trerepresentasjoner kun består av ulike plan sentrert rundt et punkt vil disse være synlige om vi ser på treet ovenfra.

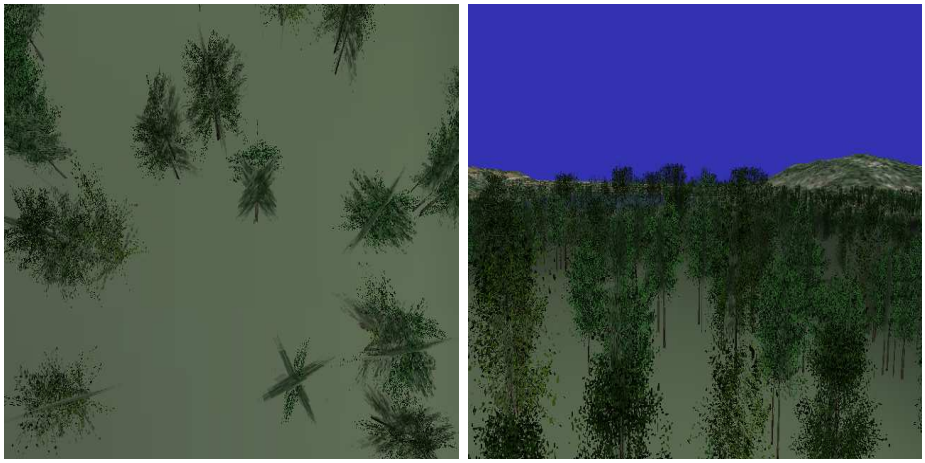
- *Befinner oss for nærme et tre*: Trerepresentasjonene gir kunstig visuell effekt om vi ser trærne fra kort avstand.

Disse to tilfellene er vist på figur 8.1.

Trær i skogmodellen vår sett ovenfra er vist til venstre på figur 8.1. Her ser vi tydelig de ulike plan som våre trerepresentasjoner består av. En mulig løsning er å innføre et nytt plan for våre trerepresentasjoner som tekstureres med et bilde av treet sett ovenfra. Vi må da ha metoder for å blende ut dette planet slik at dette ikke er synlig om vi ser treet fra siden.

På grunn av at trerepresentasjonene i denne oppgaven består av få grafiske primitiver, vil disse gi dårlig visuelt resultat om vi beveger oss for nærme dem. Det finnes en rekke artikler som omhandler visualisering av mer detaljerte trerepresentasjoner. Utfordringen blir å integrere slike modeller inn i vårt allerede eksisterende LOD hierarki.

Merk at mer detaljerte trerepresentasjoner vil nødvendigvis føre til at enda flere grafiske primitiver blir introdusert for hvert tre i skogmodellen. Dette vil igjen føre til at færre trær kan tegnes til skjerm i hver bilderamme. Et annet problem er å minimere synligheten av skift mellom de nye LOD representasjonene.



Figur 8.1: *De to tilfellene hvor våre trerepresentasjoner gir et dårlig visuelt resultat. Til venstre ser vi trerepresentasjonene ovenfra; til høyre befinner vi oss nærme trærne.*

For å øke realismen i vår skogmodell kan vi tilføre *skygger* for våre trerepresentasjoner. Et eksempel på en slik implementasjon er gitt i [4]: vi introduserer flere skyggelagte teksturer av treet avhengig av synsretningen og en lyskilde. Ulempen med denne fremgangsmåten er at flere teksturer blir introdusert for hver tretype i skogmodellen og at preprosesseringstiden for skogmodellen økes.

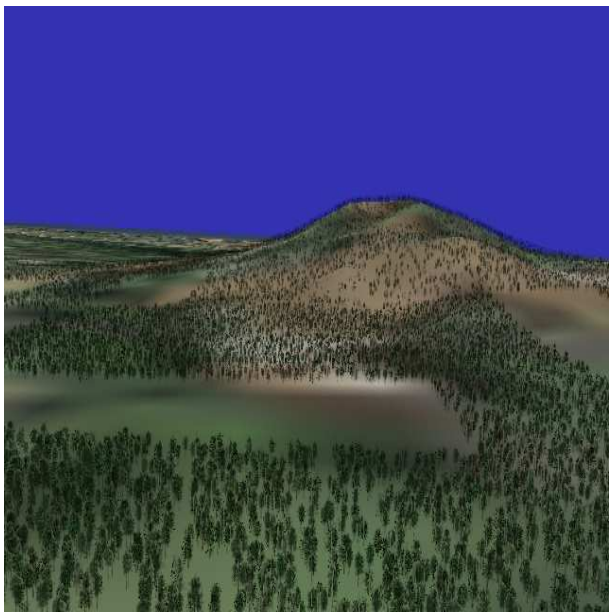
Til slutt er det ønskelig å sortere trærne i en tile etter dybde slik at alfablendingen blir korrekt. Vi nevnte hvor viktig dette var i seksjon 2.1 for at det visuelle resultatet skulle bli korrekt. I denne oppgaven har vi sett bort ifra dette fordi blendingen foregår såpass langt unna kameraet at det er vanskelig å oppdage visuelle feil. Introduserer vi mer detaljerte trerepresentasjoner, vil derimot blendingen av disse foregå såpass nærme synspunktet. Visuelle effekter vil dermed forekomme om vi ikke tar hensyn til dette.

Dette kan tenkes løst ved å sortere alle trær i en tile. I tillegg må vi tegne tiler i riktig rekkefølge i quadtreet : tilen lengst unna skal tegnes først og helt fram til tilen nærmest kamera. Oppdateringen av dybdelisten i en tile behøver kun å foregå etter at en viss toleranse med hensyn på forflytning til kameraet er nådd. Dermed slipper vi å oppdatere denne for hver bilderamme.

8.2.4 Forfining av rasteret

Ulempen med et rasterbasert datasett er at cellene fort blir synlig dersom man nærmer seg landskapet. Særlig er dette et problem for vårt tilfelle, siden trær ikke skal tegnes før man er nær terrenget. Figur 8.2 viser et eksempel på dette. Her har vi noen celler som ikke har terrengidentifikasjon skog. Dermed

skal vi heller ikke utplassere trær i disse.



Figur 8.2: *Her ser vi tydelig problemene med å bruke et rasterbasert datasett. Terrengidentifikasjonen er her forskjellig fra skog for enkelte celler i rasteret.*

Denne problemstillingen er særlig merkbar for et raster hvor oppløsningen på en celle i terrenget er stor. For vårt tilfelle spesifiserer NLCD datasettet at en celle skal spenne over et område på 30×30 meter. Gitt et rasterdatasett hvor for eksempel cellestørrelsen kun er på 2×2 meter ville ikke dette vært et så synlig problem.

En måte å løse dette på er å innføre rasterceller som kan inneholde ulik terrengidentifikasjon. En annen metode kan være å søke igjennom rasteret, identifisere slike områder og glatte dem ut ved å øke oppløsningen på hele rasteret. Uansett må vi endre på rasterdatasettet for å ta hensyn til denne problemstillingen; noe som også kan gi uønsket visuelt resultat.

Bibliografi

[1] A. Jakulin. Interactive vegetation rendering with slicing and blending. I A. de Sousa og J.C. Torres, redaktører, *Proc. Eurographics 2000 (Short Presentations)*. Eurographics, august 2000.

[2] Jitendra A. Borse og David F. McAllister. Real-time image-based rendering for stereo views of vegetation. I *Proceedings Electronic Imaging '02, San Jose, CA*, januar 2002.

[3] Inmaculada Remolar, Miguel Chover, Óscar Belmonte, José Ribelles og Cristina Rebollo. Real-time tree rendering, 2002.

[4] Alexandre Meyer, Fabrice Neyret og Pierre Poulin. Interactive rendering of trees with shading and shadows. I *Eurographics Workshop on Rendering*, juli 2001.

[5] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust og G. Turner. Real-time continuous level of detail rendering of height fields. *Proceedings of SIGGRAPH'96*, side 109–118, 1996.

[6] P. Lindstrom, D. Koller, L. F. Hodges, W. Ribarsky, N. Faust og G.Turner. Level-of-detail management for real-time rendering of photo-textured terrain. *Graphics, Visualization and Usability Center Georgia Institute of Technology Tech Report GITGVU -95-06*, 1995.

[7] Peter Lindstrom og Valerio Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. I *IEEE Transactions on Visualization and Computer Graphics, 8(3)*, side 239–254, juli-september 2002.

[8] Thomas Engh Sevaldrud. Hierarchical terrain models with applications in flight simulation. *Master Thesis, University of Oslo, Department of Informatics*, 1999.

[9] Tomas Akenine-Möller og Eric Haines. *Real-Time Rendering*. A K Peters, Ltd., 2002.

[10] David H. Eberly. *3D Game Engine Design: A practical approach to real-time computer graphics*. Academic Press, 2001.

[11] David Luebke, Martin Reddy, Jonathan D.Cohen, Amitabh Varshney, Benjamin Watson og Robert Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufman, 2003.

[12] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley Longman, Inc., 1999.

[13] William H. Press, Saul A. Teukolsky, William T. Vetterling og Bian P. Flannery. *Numerical recipes in C : the art of scientific computing*. Cambridge University Press, 2002.

[14] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr og Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. *Computer Graphics*, 32(Annual Conference Series):275–286, 1998.

[15] Mason Woo, Jackie Neider, Tom Davis og Dave Shreiner. *OpenGL Programming Guide: The official guide to learning OpenGL, Version 1.2*. Addison-Wesley, tredje utgave, 2001.

Tillegg A

Skogkonfigurasjonsfil

Som parameter inn til skogmodellen sender vi med en *skogkonfigurasjonsfil*. Denne filen gir alle nødvendige parametre som brukes i skogmodellen og leses inn under oppstarten av applikasjonen. Ved å redigere denne får man bland annet kontroll over når trær skal tegnes, hvilke trær som skal plasseres ut i landskapet og hvor dypt quadtreet skal være.

Et eksempel på en skogkonfigurasjonsfil er gitt som:

```
# NOTE: all paths are relative to the application root

# The maximum quadtree level
# (No tiles can exist on levels beneath this value)
# (value: <int>)
max_level : 7

# The maximum number of trees that can be placed in the root node
# of the quadtre
# (value: <int>)
max_trees_in_root : 500

# Our NLCD raster dataset
# (value: <string>)
dataset : data/test.asc

# The texture used for the landscape
# (value: <string>)
landscape_texture : emerald_bay.tif

# The height values (one more dimension than the raster)
# (value: <string>)
height_map : data/height.asc
```

```
# Our maximum amount of memory usage (given in megabytes)
# (value: <int>)
mem_max : 40

# The minimum level that is to be generated. All tiles from the root
# to this level are never instansiated dynamically and are never
# erased during memory cleanup
# (value: <int>)
max_pregenerated_level : 3

# For the tile error, we measure it by the help of a user-given
# constant K. This constant helps us to decide how far down a
# quadtree we need to traverse for a given epsilon measurement
# (value: <double>)
K : 0.1

# When to switch between our level of detail representations
# (in terms of epsilon)
# (value: <double>)
LOD_0 : 40
LOD_1 : 20
LOD_2 : 10

# over how long we blend in the trees (in terms of epsilon)
# (value: <double>)
transition : 9

#our tree directory where all our tree models reside
# (value: <string>)
treedir : trees

# *********** EVERGREEN TREES *****************
# --------- M_TREE4 -------------
name : M_TREE4
spread : 60
type : evergreen
height : 2.0
perturbation : 0.5

# --------- M_TREE5 -------------
name : M_TREE5
spread : 30
type : evergreen
height : 2.0
```

```
perturbation : 1.0

# ---------- M_TREE6 -----------
name : M_TREE6
spread : 10
type : evergreen
height : 1.6
perturbation : 0.7

# *********** DECIDUOUS TREES *****************
# --------- P_WILLOW tree --------
name : P_WILLOW
spread : 70
type : deciduous
height : 3.1
perturbation : 0.2

# --------- M_TREE1 --------------
name : M_TREE1
spread : 30
type : deciduous
height : 3.2
perturbation : 0.3
```

Linjer som begynner med '#' er kommentarer og vil bli hoppet over av innlesningsmetoden. Det samme gjelder tomme linjer. Parametre er gitt på følgende form:

`nøkkelord␣:␣verdi`

Ugyldig nøkkelord vil føre til terminering av programmet. Gyldige nøkkelord er de som er gitt i skogkonfigurasjonsfilen over. Det er også viktig å huske på mellomrommet både før og etter ':' for at innlesningsmetoden ikke skal gi feil verdier. Rekkefølgen på de ulike nøkkelordene bør også være som gitt i eksempelet.

Vi gir nå en oversikt over betydningen av de ulike nøkkelordene:

- *max_level* : samme som l_{maks} i denne oppgaven. Gir det maksimale nivået som en tile i quadtreet kan befinne seg på.
- *max_trees_in_root* : samme som η. Maksimale antall trær som kan utplasseres i rotnoden. Denne, samt l_{maks} bestemmer hvor mange trær som totalt blir utplassert i landskapet.

- *dataset* : Navnet på filen som inneholder NLCD rasterdatasettet. Stien til filen er gitt relativ til katalogen hvor applikasjonen kjøres fra.
- *landscape_texture* : Navnet på teksturen som skal representere landskapet.
- *height_map* : Om vi ønsker å gi høydeverdier til landskapet, må filen som inneholder høydeverdiene være gitt her.
- *mem_max* : samme som M_{maks}. Det maksimale minneforbruk som skogmodellen kan bruke før opprydning må finne sted.
- *max_pregenerated_level* : samme som l_M i denne oppgaven. Nivåer fra rota og ned til l_M bestemmer hvor mange tiles som skal pregenereres under oppstarten av applikasjonen. Kun nivåer som ligger under l_M i quadtreet kan bli slettet av opprydningsmetoden.
- K : verdien til konstanten K som er med på å beregne objektfeilen δ_t som i ligning (4.20).
- *LOD_0, LOD_1, LOD_2* : de ulike LOD_i verdiene for når vi skal bytte mellom de ulike LOD trerepresentasjonene under visualiseringen.
- *transition* : Blendinglengden *trans*.
- *treedir* : trekatalogen

Alle tremodellene vi bruker for skogmodellen må finnes i samme katalog på filsystemet. Denne katalogen kaller vi for *trekatalogen*. Skogmodellen bruker trekatalogen for å finne de ulike teksturene som et tre i landskapet vårt består av. Gitt en programrot og trekatalog, må altså treteksturene befinne seg i katalogen:

```
programrot/trekatalog/trenavn
```

For eksempel om den eksekverbare applikasjonen eksisterer i katalogen *app* og vi skal finne teksturene til et tre ved navn *willow* gitt trekatalogen *trees*, må katalogen

```
app/trees/willow
```

eksistere på filsystemet. For at skogmodellen skal finne de riktige teksturene lar vi *treteksturen* være gitt på formen:

```
programrot/trekatalog/trenavn/trenavn_billboard.tif
```

og tilsvarende for *bladteksturen*:

`programrot/trekatalog/trenavn/trenavn_leavesbillboard.tif`

Et enkelt tre i skogmodellen beskrives ved følgende variable:

`name␣:␣<verdi>`
`spread␣:␣<verdi>`
`type␣:␣<verdi>`
`height␣:␣<verdi>`
`perturbation␣:␣<verdi>`

hvor:

- *name* : Navnet på treet
- *spread* : Hvor stor andel denne tretypen dekker av skogen
- *type* : Hva slags terreng treet tilhører
- *height* : Høyden på treet
- *perturbation* : Avvik på størrelsen til treet

Vi må gi alle tretypene som tilhører barskog ("evergreen") *før* vi gir alle tretypene som tilhører løvskog("deciduous"). Alle tretypene som tilhører samme skogstype må være gitt slik at summen av *spread* verdiene for de ulike tretypene er 100. Dette skyldes metoden vi har brukt i denne oppgaven for å utplassere ulike tretyper.